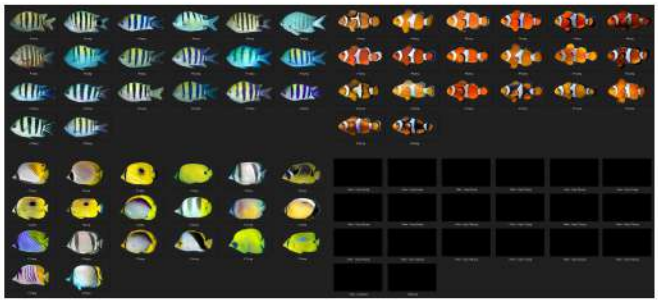
BAB IV

HASIL DAN PEMBAHASAN

4.1 *Training Strong Classifier*

4.1.1 *Input* Gambar dan *labeling*



Gambar 4.1: Gambar-gambar yang akan dipakai untuk training

Langkah pertama dalam melatih *classifier* adalah dengan memasukan *dataset* untuk latihan berserta label-labelnya. Gambar pertama dimasukan ke dalam folder sesuai dengan kelasnya. Dalam situasi ini ada empat folder yaitu, untuk kelas satu: abudefduf, untuk kelas dua: amphiprion, untuk kelas tiga: chaetodon dan terakhir untuk kelas nol: negative_examples. Gambar-gambar tersebut lalu akan dibaca menggunakan *library* CV2 yang bertugas juga untuk mengubah gambar menjadi *greyscale*. Berikut adalah *source code* `load_images()` yang digunakan untuk membaca set gambar latihan:

```
def load_images(directory):
  images=[]
  labels=[]
  for filename in os.listdir(directory):
    if filename.endswith(".png"):
      image_path = os.path.join(directory, filename)
      image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
      image = cv2.resize(image, (350, 200))
      images.append(image)
      labels.append(get_label(directory))
  return np.array(images), np.array(labels)
```

Gambar 4.2: *source code: read* gambar, labelisasi, pengubahan ke *greyscale*, dan memastikan ukuran gambar 350 x 200 piksel

```
def get_label(directory):
  # add more to add more class
  if directory == "fish_dataset\\abudefduf": return 1
  if directory == "fish_dataset\\amphiprion": return 2
  if directory == "fish_dataset\\chaetodon": return 3
  else: return 0
```

Gambar 4.3: *source code:* labelisasi gambar sesuai dengan foldernya

Untuk memanggil kedua fungsi `load_images` dan `get_label` diperlukan sebuah fungsi lainnya yang berfungsi juga untuk menggabungkan semua gambar dan label menjadi dua buah *array* `images` dan `labels`. Kedua *array* ini nantinya akan menjadi *dataset* utama dalam proses pelatihan. Berikut *source code* dari `combine_dataset` yang digunakan untuk membuat *array* `images` dan `labels`:

```
def combine_dataset():
  # load datasets from directories
  # add class to get_label first or the class will be considered a negative example
  abudefduf_images, abudefduf_labels = load_images("fish_dataset\\abudefduf")
  amphiprion_images, amphiprion_labels = load_images("fish_dataset\\amphiprion")
  chaetodon_images, chaetodon_labels = load_images("fish_dataset\\chaetodon")
  negatives_images, negatives_labels = load_images("fish_dataset\\negative_examples")

  # combining into a single dataset
  images = np.concatenate((abudefduf_images, amphiprion_images, chaetodon_images, negatives_images), axis =
0)
  labels = np.concatenate((abudefduf_labels, amphiprion_labels, chaetodon_labels, negatives_labels), axis =
0)

  return images, labels
```

Gambar 4.4: *source code: load* gambar-gambar dari folder yang bersangkutan dan menggabungkannya

4.1.2 *Generate Haar-like Features*

Untuk melakukan *generate feature* sebuah fungsi bernama `generate_features()` dipanggil dengan parameter lebar dan tinggi dari *sub-window* yang akan digunakan. Fungsi ini akan menghasilkan kurang lebih sekitar 520000 fitur berbeda untuk *sub-window* berukuran 50 x 50 piksel. Berikut *source code*-nya:

```
def generate_features(image_height, image_width):
  features = []
  features_list = ["Two Horizontal", "Two Vertical", "Four Diagonal", "Right Triangular", "Left Triangular"
, "Three Horizontal", "Three Vertical"]
  for i in features_list:
    match i:
      case "Two Horizontal" | "Two Vertical" | "Four Diagonal" | "Right Triangular" | "Left Triangular":
        feature_height = 4
        feature_width = 4
      case "Three Horizontal":
        feature_height = 4
        feature_width = 6
      case "Three Vertical":
        feature_height = 6
        feature_width = 4
    for w in range (feature_width, image_width+1, feature_width):
      for h in range (feature_height, image_height+1, feature_height):
        for x in range (0, image_width - w):
          for y in range (0, image_height - h):
            feature = (i, x, y, w, h)
            features.append(feature)
    return features
```

Gambar 4.5: *source code:* sebuah data dalam *features* memiliki semua informasi yang diperlukan untuk melakukan perhitungan nilai sebuah fitur. Tipe-tipe fitur akan menentukan rumus perhitungan fitur tersebut

4.1.3 *Calculating all features of all images*

Untuk mempermudah proses pembuatan *Decision tree* nantinya, semua fitur pada semua *sub-window*, pada semua gambar akan dihitung dan dimasukan kedalam sebuah dokumen .CSV mengunakan *library* Pandas. Fungsi yang dipakai untuk memulai proses ini adalah fungsi `write_csv()` pada Utilities.py. Berikut *source code*-nya:

```
def write_csv(images, labels, features, csv_name):
  print("starting write_csv")
  for window_num in range(3):
    temp_window_values = np.zeros((len(images), len(features)), dtype=object)
    image_ids = np.arange(len(images))

    for i in range(len(images)):
      new_data = Dataset(images[i], labels[i], features)
      if window_num == 0:
        temp_window_values[i] = new_data.window_1_features
      elif window_num == 1:
        temp_window_values[i] = new_data.window_2_features
      elif window_num == 2:
        temp_window_values[i] = new_data.window_3_features

    window_feature = {'image_ids': image_ids}
    for i in range(len(features)):
      column_name = f'win_{window_num + 1}_feature_{i}'
      window_feature[column_name] = temp_window_values[:, i]

    directory = f"Data/{csv_name}_window_{window_num}.csv"

    window_feature = pd.DataFrame(window_feature)
    window_feature.to_csv(directory, index=False)
  print("csv write complete!")
```

Gambar 4.6: *source code:* `write_csv()` mengambil semua gambar, label, fitur dan mengkalkulasi semua fitur untuk ketiga *sub-window*

Fungsi ini menghasilkan sebuah dokumen .csv untuk setiap *sub-window* dengan baris mengikuti jumlah gambar didalam *dataset* dan kolom mengikuti jumlah fitur yang ada. Maka dari itu untuk *dataset* 80 gambar akan dihasilkan sebuah tabel .csv dengan bentuk 80 x 520.000.

Fungsi ini berjalan cukup lama dengan jumlah gambar 80 buah. Penulis menghitung rata-rata waktu yang diperlukan bagi fungsi untuk membuat sebuah dokumen .csv adalah 45 menit minimum. Dan proses ini dilakukan sampai tiga kali agar dapat menciptakan tiga dokumen .csv untuk ketiga *sub-window*. Dokumen .csv ini nantinya akan diperlukan untuk proses pembuatan *decision tree*.

`Dataset` adalah sebuah *class* yang digunakan untuk menyimpan seluruh nilai fitur dari sebuah gambar sebelum dimasukan kedalam dokumen .csv . Berikut adalah bentuk *class* `Dataset` berserta fungsi bawaannya `Find_Feature_Value`:

```
class Dataset:
  def __init__(self, image, label, feature_list):
    self.image = image
    self.label = label
    self.window_1_features = self.Find_Feature_Value(image, feature_list, self.class_Window_offset_1[label
][0], self.class_Window_offset_1[label][1])
    self.window_2_features = self.Find_Feature_Value(image, feature_list, self.class_Window_offset_2[label
][0], self.class_Window_offset_2[label][1])
    self.window_3_features = self.Find_Feature_Value(image, feature_list, self.class_Window_offset_3[label
][0], self.class_Window_offset_3[label][1])

  def Find_Feature_Value(self, image, feature_list, x_offset, y_offset):
    features = np.zeros(len(feature_list), dtype=object)
    for i in range(len(feature_list)):
      feature_type, x, y, width, height = feature_list[i]
      x += x_offset
      y += y_offset
      updated_feature = (feature_type, x, y, width, height)
      data_features = compute_feature_with_matrix(image, 0, updated_feature)
      features[i] = data_features
    return features
```

Gambar 4.7: *source code: class Dataset*

`Dataset`, untuk melakukan perhitungan fitur, digunakan *Offset* menspesifikasi lokasi mulut, sirip dan ekor ikan pada gambar. Hal ini dilakukan karena masing-masing kelas ikan semuanya memiliki mulut, sirip dan ekor dengan lokasi yang berbeda satu dengan yang lainnya. Contohnya: Mulut dari genus ikan Chaetodon agak lebih kebawah dibandingkan kedua kelas ikan lainnya. Selain itu juga untuk setiap bagian ikan yang akan diklasifikasi akan dicari lokasi yang paling terlihat unik daripada kelas-kelas lainnya. Contohnya pada kelas genus ikan Abudefduf diambil bagian ekor yang memberntuk huruf "V" sementara pada kelas genus Amphiprion bagian kelas yang dipelajari adalah bagian melengkung bagian atas diantara ekor dan badan. Berikut adalah *offset* yang penulis gunakan untuk pelatihan:

```
    class_Window_offset_1 = [
      # order according to label's order in LoadImages
      # for searching mouth features
      (0, 0),
      (0, 88),
      (0, 73),
      (15, 100)
    ]

    class_Window_offset_2 = [
      # order according to label's order in LoadImages
      # for searching fin features
      (0, 0),
      (116, 0),
      (153, 9),
      (116, 0)

    ]

    class_Window_offset_3 = [
      # order according to label's order in LoadImages
      # for searching tail feature
      (0, 0),
      (280, 89),
      (237, 47),
      (277, 80)
  ]
```

Gambar 4.8: *source code: offset* ini di-inisalisasi untuk setiap kelas Dataset sehingga bisa diakses langsung oleh fungsi `Find_Feature_Value`

Untuk perhitungan nilai fitur dari gambar digunakan fungsi `compute_feature_with_matrix()`. Pertama data fitur diubah dengan menambah lokasi x dan y dari fitur dengan `x_offset`, dan `y_offset` lalu fitur akan mengembalikan sebuah nilai float dari hasil perhitungan tersebut. Berikut *source code* dari `compute_feature_with_matrix`:

```
def compute_feature_with_matrix(image, feature):
  feature_type, x, y, width, height = feature
  # +1 due to slicing paramter = start at:stop before
  match feature_type:
    case "Two Horizontal":
      white = np.sum(image[y:y + height + 1, x:x + int(width/2) + 1])
      black = np.sum(image[y:y + height + 1, x + int(width/2):x + width + 1])
    case "Two Vertical":
      white = np.sum(image[y:y + int(height/2) + 1, x:x + width+1])
      black = np.sum(image[y + int(height/2):y + height + 1, x:x + width+1])
    case "Three Horizontal":
      white = np.sum(image[y: y + height + 1, x:x + int(width/3) + 1]) + np.sum(image[y: y + height + 1, x
+ int(width*2/3):x + width + 1])
      black = np.sum(image[y: y + height + 1, x + int(width/3):x + int(width*2/3) + 1])
    case "Three Vertical":
      white = np.sum(image[y:y + int(height/3) + 1, x:x + width + 1]) + np.sum(image[y + int(height*2/3):y
+ height + 1, x: x + width + 1])
      black = np.sum(image[y + int(height/3):y + int(height*2/3) + 1, x:x + width + 1])
    case "Four Diagonal":
      white = np.sum(image[y:y + int(height/2) + 1, x + int(width/2): x + width + 1]) + np.sum(image[y +
int(height/2):y + height + 1, x: x + int(width/2) + 1])
      black = np.sum(image[y:y + int(height/2) + 1, x:x + int(width/2) + 1]) + np.sum(image[y + int(height
/2): y + height + 1, x + int(width/2):x + width + 1])
    case "Right Triangular":
      matrix = image[y:y + height + 1, x:x + width + 1]
      white = np.sum(np.tril(matrix))
      black = np.sum(np.triu(matrix))
    case "Left Triangular":
      matrix = np.rot90(image[y:y + height + 1, x:x + width + 1], k=3)
      white = np.sum(np.tril(matrix))
      black = np.sum(np.triu(matrix))
  return int(white) - int(black)
```

Gambar 4.9: *source code: offset* ini di-inisalisasi untuk setiap kelas Dataset sehingga bisa diakses langsung oleh fungsi `Find_Feature_Value()`

4.1.4 *Create Decision Tree for each Feature*

Setelah semua nilai fitur sudah dihitung dan dimasukan kedalam dokumen .csv, selanjutnya bisa dimulai proses pembuatan *decision tree* atau *weak classifier*. Proses ini dimulai dengan pertama membagi contoh menjadi tiga kelompok, yaitu data *training*, data *testing* dan data *validation*. Hal ini dilakukan dengan menggunakan fungsi `split_data` dibantu dengan fungsi `train_test_split()` dari *library* sklearn:

```
def split_data(features, csv_name, labels):
  data = DecisionTree.get_data(features, csv_name)
  labels_df = pd.DataFrame({'Label' : labels})
  data = pd.concat([data, labels_df], axis=1)

  X = data.iloc[:, :-1].values
  Y = data.iloc[:, -1].values.reshape(-1, 1)

  X_temp, X_train, Y_temp, Y_train = train_test_split(X, Y, test_size=0.3, random_state=42)
  X_valid, X_test, Y_valid, Y_test = train_test_split(X_temp, Y_temp, test_size=0.5, random_state=42)

  print(type(X_train))
  splits = [X_train, Y_train, X_test, Y_test, X_valid, Y_valid]
  return splits
```

Gambar 4.10: *source code:* data *training*, data *testing* dan data *validation* disimpan kedalam *array* `X_train, Y_train, X_test, Y_test, X_valid, Y_valid`. Yang lalu disimpan kedalam `splits`

Label baru ditambahkan sekarang agar tidak menggangu proses penulisan kolom .csv yang dinamis mengikuti jumlah fitur yang ada. Hasilnya adalah sebuah *object* bernama `splits` yang memiliki *dataframe*: `X_train, Y_train, X_test, Y_test, X_valid, Y_valid`. *Dataframe* dengan data awalan X berisikan nilai-nilai fitur yang sudah dikalkulasi di tahap sebelumnya. Sementara data awalan Y berisikan label untuk data X.

`split_data()` mengambil data dari .csv menggunakan fungsi bernama `get_data`. Fungsi ini hanya bertugas untuk membaca .csv saja dengan bantuan fungsi `read_csv` dari *library* Pandas:

```
def get_data(features, csv_name):
  col_names = ['image_ids']
  for i in range(len(features)):
    temp_column_name = f'win_1_feature_{i}'
    col_names.append(temp_column_name)
  return Utilities.read_csv(csv_name, col_names)

def read_csv(csv_name, col_names):
  # used to read all column
  directory = "Data/" + csv_name + ".csv"
  data = pd.read_csv(directory, skiprows=1, header=None, names = col_names)
  return data
```

Gambar 4.11: *source code:* get data dan readcsv yang digunakan oleh split data

setelah data di-*split*, barulah *decision tree* bisa deibuat dengan menggunakan fungsi `build_all_tree`:

```
def build_all_tree(splits, features):
  classifiers = [None] * len(features)
  classifiers_accuracy = [0] * len(features)
  X_train, Y_train, X_test, Y_test, X_valid, Y_valid = splits
  minimum_splits = 3
  maximum_depth = 3
  for i in range(len(features)):
    if i % 1000 == 0: print (f'starting tree {i}')
    classifier = DecisionTreeClassifier(minimum_splits, maximum_depth)
    classifier.fit(temp_X_train, Y_train)
    # classifier.print_tree()

    classifiers[i] = classifier
    Y_pred = classifier.predict(X_test)
    classifiers_accuracy[i] = accuracy_score(Y_test, Y_pred)
  return classifiers, classifiers_accuracy
```

Gambar 4.12: *source code:* `build_all_tree` untuk membuat semua decision tree untuk setiap fitur

Pada tahap ini `build_all_tree` mengisolasi kolom nilai fitur yang berhubungan dan menyimpannya pada `temp_X_train`, yang nantinya akan digunakan saat pembuatan *decision tree*. *Decision tree* di-inisialisais membuat *class* `DecisionTreeClassifier` dan disimpan menjadi `classifier`. `Classifier` lalu dilatih menggunakan fungsi `fit()`. Hasil dari pelatihan ini lalu langsung dites menggunakan fungsi `accuracy_score()` dari *library* Sklearn. Variabel `classifiers_accuracy` ini nantinya akan dipakai dalam proses *Boosting*. Untuk keseluruhan *class* `DecisionTreeClassifier` dan fungsi-fungsinya bisa dilihat berkut ini:

```
class Node():
  def __init__(self, feature_index=None, threshold=None, left=None, right=None, info_gain=None, value=
None):
    self.feature_index = feature_index
    self.threshold = threshold
    self.left = left
    self.right = right
    self.info_gain = info_gain

    self.value = value
```

Gambar 4.13: *source code: class* `node` digunakan untuk menyimpan informasi cabang dan *threshold* pada *node decision tree*

```
class DecisionTreeClassifier():
  def __init__(self, minimum_splits = 2, maximum_depth = 2):
    self.root = None

    # stopping condition
    self.minimum_splits = minimum_splits
    self.maximum_depth = maximum_depth
```

Gambar 4.14: *source code: class* digunakan untuk menyimpan tinggi maksimal dan minimal *split* pada *decision tree*. Semua data lainnya disimpan pada `node`

```
def build_tree(self, training_dataset, current_depth = 0):

  X, Y = training_dataset[:,:-1], training_dataset[:,-1]
  num_samples, num_features = np.shape(X)

  # split until conditons are met
  if num_samples >= self.minimum_splits and current_depth <= self.maximum_depth:
    # find best split
    best_split = self.get_best_split(training_dataset, num_features)
    # check if information gain is positive
    if best_split["info_gain"]>0:
      left_subtree = self.build_tree(best_split["dataset_left"], current_depth+1)
      right_subtree = self.build_tree(best_split["dataset_right"], current_depth+1)
      return Node(best_split["feature_index"], best_split["threshold"], left_subtree, right_subtree,
best_split["info_gain"])

  leaf_value = self.calculate_leaf_value(Y)
  return Node(value=leaf_value)
```

Gambar 4.15: *source code:* fungsi utama dari *class* DecisionTreeClassifier

Fungsi `build_tree()` adalah fungsi utama untuk pelatihan *decision tree* yang berjalan secara rekursif sampai sebuah daun sudah didapat, atau kedalaman maksimum sudah dicapai. Sebuah daun sudah didapat bilamana `info gain` dari fungsi `get_best_split` adalah 0, atau *node* sudah tidak perlu dipecah lagi karena mencapai potensi maksimum *decision tree*.

```
def get_best_split(self, dataset, num_features):
  # dictionary to save data
  best_split = {
  "info_gain": -float("inf")  # Initialize info_gain to a very small value
  }
  max_info_gain = -float("inf")

  for feature_index in range(num_features):
    feature_values = dataset[:, feature_index]
    potential_thresholds = np.unique(feature_values)

    for threshold in potential_thresholds:
      # get curent split
      dataset_left, dataset_right = self.split(dataset, feature_index, threshold)
      # check if child not null
      if len(dataset_left) > 0 and len(dataset_right) > 0:
        y, left_y, rigth_y = dataset[:,-1], dataset_left[:,-1], dataset_right[:, -1]
        # compute information gain
        current_info_gain = self.information_gain(y, left_y, rigth_y, "gini")
        # update best split if needed
        if current_info_gain > max_info_gain:
          best_split["feature_index"] = feature_index
          best_split["threshold"] = threshold
          best_split["dataset_left"] = dataset_left
          best_split["dataset_right"] = dataset_right
          best_split["info_gain"] = current_info_gain
          max_info_gain = current_info_gain

      return best_split
```

Gambar 4.16: *source code:* fungsi get_best_split

Fungsi `get_best_split()` bertugas untuk mencari *threshold* paling sesuai untuk memecah cabang suatu *node* dengan mengetes satu-persatu nilai atribut dari data latihan. Atribut disini adalah nilai *feature* yang sedang dilatih dari semua gambar dari set *train*. Untuk mencari *info gain*, *gini* akan dihitung menggunakan fungsi `information_gain`. Selain *gini*, menghitung `information gain` juga dapat dilakukan dengan menggunakan *entropy*.

```
def split(self, dataset, feature_index, threshold):
  # fuction to split data
  dataset_left = np.array([row for row in dataset if row[feature_index] <= threshold])
  dataset_rigth = np.array([row for row in dataset if row[feature_index] > threshold])
  return dataset_left, dataset_rigth
```

Gambar 4.17: *source code:* fungsi split hanya bertugas membagi `node` berdasarkan *threshold* yang sudah ditemukan

```
    def information_gain(self, parent, left_child, right_child, mode="entropy"):
      weight_left = len(left_child) / len(parent)
      weight_rigth = len(right_child) / len(parent)
      if mode == "gini":
        gain = self.gini_index(parent) - (weight_left * self.gini_index(left_child) + weight_rigth * self.
gini_index(right_child))
      else:
        gain = self.entropy(parent) - (weight_left * self.entropy(left_child) + weight_rigth * self.entropy
(right_child))
      return gain
```

Gambar 4.18: *source code:* `information_gain()` mencari data dengan menghitung *gini* atau *entropy*

```
    def entropy(self, y):
      # fuction to count entropy
      class_labels = np.unique(y)
      entropy = 0
      for cls in class_labels:
        p_cls = len(y[y == cls]) / len(y)
        entropy += -p_cls * np.log2(p_cls)
      return entropy
```

Gambar 4.19: *source code:* perhitungan *entrophy*

```
    def gini_index(self, y):
      # function to count gini index (lebih cepet aja karna gak pake log)
      class_labels = np.unique(y)
      gini = 0
      for cls in class_labels:
        p_cls = len(y[y == cls]) / len(y)
        gini += p_cls**2
      return 1 - gini
```

Gambar 4.20: *source code:* perhitungan *gini*

```
    def calculate_leaf_value(self, Y):

      Y = list(Y)
      return max(Y, key = Y.count)
```

Gambar 4.21: *source code:* fungsi untuk mencari mayoritas kelas pada *leaf node*

```
def fit(self, X, Y):
  # fuction to train tree
  dataset = np.concatenate((X, Y), axis = 1)
  self.root = self.build_tree(dataset)
```

Gambar 4.22: *source code:* fungsi `fit()` adalah fungsi yang dipanggil untuk mulai membangun *decision tree* setelah dibuat

Terakhir, fungsi `Predict()` akan digunakan untuk klasifikasi yang sebenarnya. Dalam fugnsi ini fit mengambil X atau *dataset* `X_test` untuk mengetes akurasi dari *decision tree* tersebut yang lalu akan dikomparasi dengan `Y_test` menggunakan fungsi sklearn `accurac_score()`. Pencarian `accurac_score()` setiap *decision tree* disini dilakukan untuk mempercepat proses *boosting* di tahap berikutnya karena *decision tree* dapat langsung diurutkan dari yang terkuat ke yang terlemah.

```
def predict(self, X):
  # fuction to predict new dataset
  predictions = [self.make_prediction(x, self.root) for x in X]
  return predictions

def make_prediction(self, x, tree):
  # fuction to detect single datapoint
  if tree.value!=None: return tree.value
  feature_val = x[tree.feature_index]
  if feature_val <= tree.threshold: return self.make_prediction(x, tree.left)
  else: return self.make_prediction(x, tree.right)
```

Gambar 4.23: *source code:* `Predict()` digunakan untuk melakukan prediksi dengan *decision tree* yang sudah dibuat

Setelah semua *decision tree* dan akurasinya sudah dicari dan disimpan kedalam *array* classifiers dan classifiers_accuracy. Keduanya akan disimpan kedalam dokumen Pickle untuk direferensi kedepannya. Penyimpanan kedalam dokumen pickle ini bertujuan agar proses pelatihan tidak perlu diulangi berulang kali bila ada masalah di tahapan berikutnya. Berikut *source code* penyimpanan *decision tree* kedalam Pickle:

```
class PickleTree:
  def __init__(self, features, trees, accuracies):
    self.feature_num = np.arange(len(features))
    self.trees = trees
    self.accuracies = accuracies

def dump_to_pickle(file_name, object):
  directory = "Data/" + file_name + ".pickle"
  with open(directory, 'wb') as file:
    pickle.dump(object, file)
```

Gambar 4.24: *source code:* penyimpanan *decision tree* kedalam pickle

Untuk menyimpan *decision tree* kedalam pickle, pertama semua *decision tree* dan akurasinya disimpan kedalam *class* bernama `PickleTree` yang lalu akan di *dump* menggunakan fungsi `dunp_to_pickle()` kedalam *directory* yang sudah ditentukan.

4.1.5 *Boosting*

Setelah dokumen pickle dari semua *decision tree* atau *weak classifier* dibuat. Semua *weak classifier* akan di-*boosting* untuk memberikan bobot *voting* untuk semuanya. Hal ini dilakukan dengan mengetes *weak classifier* secara berurutan dari yang terkuat ke yang terlemah. Contoh-contoh latihan yang sulit untuk diklasifikasi *weak learner* sebelumnya akan diberikan nilai lebih bila berhasil diklasifikasi *weak learner* berikutnya. Proses ini kita mulai dengan memanggil fungsi `training_strong_classifier()`:

```
def training_strong_classifier(features, trees, splits, accuracy, pickle_name):
  X_train, Y_train, X_test, Y_test, X_valid, Y_valid = splits
  image_weights = Boosting.initialize_weight(Y_test)
  print(np.sum(image_weights))
  orderlist = np.arange(len(accuracy))
  orderlist = Boosting.get_initial_sorted_accuracy(accuracy, orderlist)

  initial_accuracy = float('-inf')
  current_accuracy = 0
  iteration = 0
  limit = 100 #change according to needs

  # start boosting loop. Will stop when accuracy fell or iteration hit limit
  while True:
    alpha_list = Boosting.start_boosting(trees, X_test, Y_test, image_weights, orderlist)
    validation_prediction = Boosting.strong_prediction(trees, orderlist, X_valid, alpha_list)

    initial_accuracy = current_accuracy
    print(f'current initial accuracy: {initial_accuracy}')
    current_accuracy = accuracy_score(Y_valid, validation_prediction)
    print(f'current after boosting accuracy: {current_accuracy}')

    # check whether accruacy deteriorate or limit hit
    if current_accuracy <= initial_accuracy or iteration >= limit:
      print('final accuracy deteriorate, rolling back to last iteration...')
      alpha_list = last_iteration_alpha_list
      orderlist = last_iteration_orderlist
      break

    print('starting over. Saving alpha...')
    alpha_list, orderlist = Boosting.get_sorted_accuracy(alpha_list, orderlist)
    last_iteration_alpha_list = alpha_list
    last_iteration_orderlist = orderlist
    iteration += 1


  # saving trees, related features and its order in pickle
  final_trees = np.empty(len(orderlist), dtype=object)
  final_features = np.empty(len(orderlist), dtype=object)
  for i in range(len(orderlist)):
    final_trees[i] = trees[orderlist[i]]
    final_features[i] = features[orderlist[i]]

  pickle_this = PickleTreeFinal(final_features, final_trees, alpha_list)
  Utilities.dump_to_pickle(f'{pickle_name}', pickle_this)
```

Gambar 4.25: *source code:* training_strong_classifier

Pertama dalam fungsi ini harus dicari bobot nilai dari setiap contoh latihan, bobot nilai ini berbeda dari bobot *voting weak learner*. Fungsi bobot nilai adalah menaikan kuatnya *voting* sebuah *weak classifier* bila *weak learner* berhasil mengklasifikasi sebuah contoh latihan dengan benar, oleh karena itu jumlah nilai total dari bobot latihan atau image_weights haruslah kurang lebih satu. `image_weights` di-inisialisasi menggunakan fungsi `initialize_weights()` berikut:

```
def initialize_weight(test_images):
  image_weights = np.ones(len(test_images)) / len(test_images)
  return image_weights
```

Gambar 4.26: *source code:* inisialisasi bobot gambar sebelum /emphBoosting

Bisa dilihat proses penghitungan bobot dari `image_weights` hanyalah pembagian satu dengan jumlah total contoh latihan (disini len(`test_images`)). Untuk proses pelatihan menggunakan 80 contoh gambar latihan, fungsi `split()` telah mengalokasikan 28 contoh untuk digunakan dalam tahap *Boosting*, yang disimpan dalam `X_valid` dan `Y_valid`. Berikutnya fungsi `get_initial_sorted_accuracy()` dipanggil untuk mengurutkan *weak classifier* berdasarkan akurasi yang sudah didapat pada tahap sebelumnya:

```
def get_initial_sorted_accuracy(accuracy, orderlist):
  accuracy_threshold = 0.4
  accuracy, orderlist = zip(*sorted(zip(accuracy, orderlist), reverse = True))
  orderlist = [classifier for accuracy, classifier in zip(accuracy, orderlist) if accuracy >=
accuracy_threshold]
  return orderlist
```

Gambar 4.27: *source code:* pengurutan *weak classifier* berdasarkan akurasi

Pada tahap *sorting* ini juga dilakukan elimanasi *weak classifier* yang terlalu lemah. Awalnya penulis mengeliminasi *weak classifier* yang memiliki nilai akurasi dibawah 50% namun karena takut jumlah *weak classifier* terlalu sedikit, maka penulis menurunkan *threshold* menjadi 40%. Eliminasi ini secara signifikan mengurangi jumlah *classifier* yang awalnya berjumlah sekitar 520.000 menjadi: 6742 *weak classifier* pada *classifier* jendela kiri, 8231 *weak classifier* pada *classifier* jendela tengah, dan 10588 *weak classifier* pada *classifier* jendela kanan. Eliminasi yang besar ini mengimplikasikan bahwa mayoritas *weak classifier* yang dibuat dengan mencoba semua probabilitas yang ada memiliki akurasi dibawah 40% dan mungkin hanya akan berkontribusi saja kepada klasifikasi akhir. Selanjutnya proses *Boosting dilanjutkan* dengan mencari alpha_list atau bobot voting setiap *weak classifier* menggunakan fungsi start_boosting(). Berikut adalah *source code*-nya:

```
def start_boosting(trees, X_test, Y_test, image_weights, orderlist):
  print('Boosting...')
  alpha_list = np.zeros(len(orderlist))
  for i in range(len(orderlist)):
    # make prediction with i-th tree
    treeN = orderlist[i]
    prediction  = trees[treeN].predict(X_test)

    # calculate error of the tree
    indicator = np.array(np.array(prediction).astype(int) != Y_test.flatten(), dtype = float)
    epsilon = np.sum(image_weights * indicator) / np.sum(image_weights)

    # calculate the weight of the tree
    alpha = 0.5 * np.log((1 - epsilon) / (epsilon + 1e-10)) + np.log(4 - 1) #1e-10 const added to prevent
div by 0. 4 is number of class
    if alpha < 1e-10: alpha = 1e-10 #1e-10 const added to prevent alpha getting too small in np.exp(alpha
* indicator) later
    alpha_list[i] = alpha

    # update the weight for the samples so the sum of image_weight will be close to 1 for the next
iteration
    image_weights *= np.exp(alpha * indicator)
    image_weights /= np.sum(image_weights)

  return alpha_list
```

Gambar 4.28: *source code:* pencarian nilai bobot *boosting* menggunakan fungsi `start_boosting()`

Disini hasil klasifikasi yang dilakukan oleh *weak classifier* akan dibandingkan dengan label aslinya yang tersimpan di `Y_valid` dan disimpan pada array `indicator` dalam nilai 0 bila klasifikasi dilakukan secara benar, dan 1 bila klasifikasi dilakukan secara salah. Kemudian `alpha` atau bobot voting sang *weak classifier* akan dihitung. Para perhitungan ini, `epsilon` akan ditambahkan dengan 1e-10 untuk mencegah pembagian dengan angka 0 bilamana *weak classifier* benar mengklasifikasi semua contoh dan menghasilnya `indicator` yang hanya berisi angka 0 saja. np.log(4 - 1) disini digunakan agar `alpha` tidak negatif, 4 pada formula ini adalah jumlah kelas yang sedang diklasifikasi yaitu kelas negatif, Abudefduf, Amphiprion, dan Chaetodon. Berikutnya `alpha` dicek supaya tidak lebih kecil daripada 1e-10 agar tidak menyebabkan normalisasi bobot gambar yang salah di bagian berikutnya. terakhir `image_weight` diupdate, dimana gambar yang salah diklasifikasi akan dinaikan nilainya, baru setelahnya nilai dinormalisasi lagi agar kurang lebih berjumlah 1. Setelah bobot *voting* sudah dicari, seluruh *weak learner* pada tahap ini akan dites layaknya klasifikasi yang sebeneranya, dimana nilai *voting* setiap *weak classifier* akan diperhitungkan untuk memilih hasil klasifikasi. Klasifikasi pada tahap ini dilakukan oleh fungsi

`strong_prediciton`:

```
def strong_prediction(trees, orderlist, X_valid, alpha_list):
  predictions = [0] * len(X_valid)
  scoreboard = [[0, 0, 0, 0] for _ in range(len(X_valid))]
  for i in range(len(orderlist)):
    tree_index = orderlist[i]
    prediction = trees[tree_index].predict(X_valid)

    # add score to scoreboard according to results and alpha value of tree
    for j in range(len(prediction)):
      weak_learner_prediction = int(prediction[j])
      scoreboard[j][weak_learner_prediction] += 1 * alpha_list[i]

  # return score to the main scoreboard
  for k in range(len(prediction)):
      # print(f'scoreboard {k}: {scoreboard[k]}')
      predictions[k] = scoreboard[k].index(max(scoreboard[k]))
  return predictions
```

Gambar 4.29: *source code:* klasifikasi yang dilakukan setelah setiap iterasi *boosting*

Saat klasifikasi. akan dibuatkan variabel `scoreboard` untuk mencatat total nilai voting setiap kelas. Contohnya suatu *weak classifier* dengan bobot *voting* 0.67 mengklasifkasi suatu fitur sebagai kelas 1 atau Abudefduf. maka *scoreboard* akan berubah manjadi [0, 0.67, 0, 0]. Lalu misalnya *weak classifier* lain dengan bobot *voting* 0.2 memilih kelas 3 atau Amphiprion, maka *scoreboard* akan menjadi [0, 0.67, 0.2, 0]. Klasifikasi akan diakhiri ketika semua *weak learner* sudah dipakai. Setelah itu kelas dengan nilai *voting* paling tinggi akan dipilih sebagai hasil dari klasifikasi. Yang lalu akan dibandingkan dengan `Y_valid` untuk dicari akurasinya.

Proses iterasi *boositng* ini akan diulang terus menerus hingga tingkat akurasi klasfikasi menggunakan *weak classifier* berbobot mulai mengalami penurunan. Dalam situasi ini nilai bobot *voting* dan urutan *voting* pada iterasi sebelumnya akan diambil dan disimpan kedalam dokumen pickle, kali ini dalam *class* `PickleTreeFinal` dengan fungsi `dump_to_pickle()` sebelumnya:

```
class PickleTreeFinal:
  def __init__(self, features, trees, alpha_list):
    self.features = features
    self.trees = trees
    self.alpha_list = alpha_list
```

Gambar 4.30: *source code:* bentuk *class PickleTreeFinal*

Perbedaan pickle ini dengan pickle yang menyimpan seluruh fungsi pada tahap sebelumnya adalah pada pickle ini juga disimpan info *feature* juga sesuai dengan urutan dari *weak classifier* yang berhubungan. Hal ini dilakukan agar pada tahapan berikutnya, *cascade* dapat langsung digunakan untuk klasifikasi sebenarnya, yang memerlukan info *features* untuk dapat langsung membaca nilai fitur langsung dari gambar. Hasil dari *boosting* untuk ketiga *strong classifier* cukup memuaskan dengan akurasi 71% untuk ketiga *strong classifier*.

Tabel 4.1: Hasil boosting pada ketiga *window*

Window pickle name	**Num. Features**	**Accuracy**	**Top Feature**
window_0_strong_classsifier	2751	71%	('Right Triangular', 26, 24, 4, 4)
window_1_strong_classsifier	4521	71%	('Four Diagonal', 4, 40, 8, 4)
window_2_strong_classsifier	6749	71%	('Two Horizontal', 0, 0, 4, 4)

4.1.6 *Training Cascade*

Pelatihan pickle dimulai dengan pertama membuat *class* `Cascade` yang nantinya akan diisi dengan *class* `stage` yang berisikan *weak classsifier* dengan bobot voting mereka. Berikut bentuk *class* Cascade:

```
class Cascade:

  def __init__(self):
    self.stages = []
```

Gambar 4.31: *source code:* bentuk *class Cascade*

Selanjutnya `stages` akan diisi dengan fungsi `fill_cascade()` dengan *source code* sebagai berikut:

```
def fill_cascade(self, features, trees, alpha_list, splits):
  print(f'starting to fill cascade...')
  X_train, Y_train, X_test, Y_test, X_valid, Y_valid = splits
  used_features = 0
  print(f'number of used_features: {used_features}')
  while True:
    if used_features >= len(features): break
    new_cascade = CascadeStage()
    new_cascade.train_stage(features, trees, alpha_list, X_valid, Y_valid, used_features)
    used_features += len(new_cascade.trees) #check the total number of features used
    self.stages.append(new_cascade)
  print(f'cascade is finished!')
```

Gambar 4.32: *source code:* fungsi untuk mengisi `stages` pada *Cascade*

Fungsi `fill_cascade` disini mengambil `features`, `trees` dan `alpha_list` yang sudah diurutkan dan di-*boosting* pada tahap sebelumya. Lalu sebuah *class* baru dibuat untuk menyimpan ketiganya pada sebuah *stage*, *class* itu adalah `CascadeStage` dengan *source code* sebagai berikut:

```
class CascadeStage:
  def __init__(self):
    self.features = []
    self.trees = []
    self.alpha_list = []
```

Gambar 4.33: *source code: class* CascadeStage

yang lalu diisi dengan menggunakan fungsi `train_stage()` sebagai berikut:

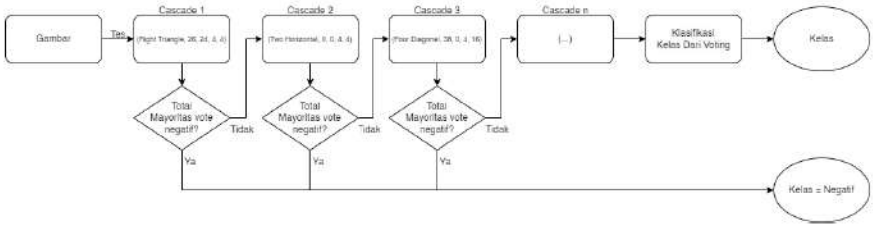
```
def train_stage(self, features, trees, alpha_list, X_valid, Y_valid, used_features):
  detection_rate = 0
  while detection_rate < 0.5:
    if used_features >= len(features): break
    # append weak classifier into stage one by one
    self.features.append(features[used_features])
    self.trees.append(trees[used_features])
    self.alpha_list.append(alpha_list[used_features])

    orderlist = np.arange(len(self.trees))
    validation_prediction = Boosting.strong_prediction(self.trees, orderlist, X_valid, self.alpha_list)
    detection_rate = accuracy_score(Y_valid, validation_prediction)
    used_features += 1
  print(f'features used in this stage: {used_features}')
```

Gambar 4.34: *source code:* pelatihan *stage* dalam *Cascade*

Pada fungsi ini *weak classifier* ditambah satu persatu berserta bobot dan *feature*-nya, lalu *stage* akan dites menggunakan fungsi `strong_prediction` yang dipakai juga di tahap *boosting*. Penggunaan fungsi ini dapat dilakukan karena struktur dan cara klasifikasi dari *stage* mirip dengan sederet *weak classifier* pada tahap sebelumnya. Iterasi ini lalu diteruskan hingga akurasi *stage* mencapai 50% atau sampai *weak classifier* habis. Setelah itu *stage* yang sudah dibuat akan di-`append()` kedalam *array stages* dan proses yang sama diulangi hingga semua *weak classifer* sudah habis terpakai. Terakhir *Cascade* lalu disimpan kedalam pickle untuk digunakan dalam klasifikasi yang sebenarnya. Berikut fungsi yang digunakan untuk menyimpan *cascade* kedalam pickle:

```
def save_to_pickle(self, pickle_name):
  print(f'saving to Pickle...')
  Utilities.dump_to_pickle(f'{pickle_name}', self)
  print(f'complete!')
```

Gambar 4.35: *source code:* fungsi penyimpanan *Cascade* ke pickle dengan menggunakan fungsi `dump_to_pickle` lagi



Gambar 4.36: Gambaran *cascade* window 1 (kiri) dan cara kerjanya

4.2 Validasi

Untuk proses validasi atau penggunaan, penulis telah membuat sebuah *file* python baru untuk mengklasifikasi menggunakan *cascade* yang telah dibuat. Untuk gambar yang akan diklasifikasi harus memiliki ukuran 350 x 200 piksel, bertipekan *Portable Network Graphics* atau PNG, dan dengan latar belakang sudah dihilangkan. Gambar-gambar yang akan diklasifikasi harus dimasukan kedalam *folder* bernama *classification_target*, dan pengguna juga membuat satu *folder* lain bernama *classification_results* untuk hasil klasifikasi. Totalnya penulis telah mengumpulkan

75 gambar ikan baru dengan jumlah tiap kelas 25 gambar. Berikut *source code* dari predict.py:

```
import numpy as np
import os
import cv2
from Cascade import *
from Utilities import *

# load cascades for each window
window_cascade = [None, None, None]
window_prediction = np.zeros(3)
window_cascade[0] = Utilities.read_from_pickle('window_0_cascade') #for left side/mouth detection
window_cascade[1] = Utilities.read_from_pickle('window_1_cascade') #for mid side/fin detection
window_cascade[2] = Utilities.read_from_pickle('window_2_cascade') #for right side/tail detection

directory = "classification_target"

for filename in os.listdir(directory):
  if filename.endswith(".png"):
    image_path = os.path.join(directory, filename)
    image_name = filename

    #load target image for classification
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    image_unedited = cv2.imread(image_path, cv2.IMREAD_UNCHANGED)
    image = cv2.resize(image, (350, 200))
    image_width = 350
    image_height = 200

    # scan for the whole image using sliding windows
    for i in range(len(window_cascade)):
      # offset for different part all 3 window
      match i:
        case 0: left_window_width = 0
        case 1: left_window_width = int(image_width / 3)
        case 2: left_window_width = int(image_width / 3 * 2)
      for y in range(0, image_height - 50 + 1):
        for x in range(0, int(image_width / 3) - 50 +1):
          prediction = window_cascade[0].final_cascade_classification(image, x + left_window_width, y)
          if prediction != 0: break

        if prediction != 0: break
      # print(f'classification result for window {i}: {prediction}')
      window_prediction[i] = prediction

    # count majority vote and predict class
    print(f'result of {image_name} classification: {window_prediction}')
    unique_elements, counts = np.unique(window_prediction, return_counts=True)
    max_count_index = np.argmax(counts)

    if counts[max_count_index] > len(window_prediction) // 2:
      image_class = unique_elements[max_count_index]
    else:
      image_class = 0

    match image_class:
      case 0: image_class = 'None'
      case 1: image_class = 'Abudefduf'
      case 2: image_class = 'Amphiprion'
      case 3: image_class = 'Chaetodon'
```

Gambar 4.37: *source code:* predict.py untuk melakukan klasifikasi sebenarnya (bagian 1)

```
            position = (10, 30)
            font = cv2.FONT_HERSHEY_SIMPLEX
            font_scale = 1
            font_thickness = 2
            font_color = (0, 0, 255)

            output_image_path = os.path.join('classification_results\\', os.path.splitext(filename)[0] + '.jpg'
    )
            cv2.putText(image_unedited, image_class, position, font, font_scale, font_color, font_thickness)
            cv2.imwrite(output_image_path, image_unedited)
            print('anotated image completed!')
```

Gambar 4.38: *source code:* predict.py untuk melakukan klasifikasi sebenarnya (bagian 2)

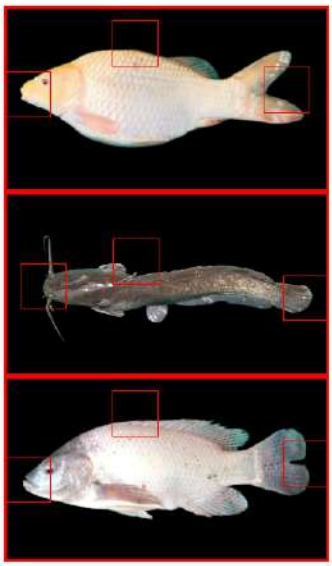
Gambar 4.39: Ketiga kelas ikan

`Predict.py` pertama membaca gambar-gambar yang akan diklasifikasi dari folder *classification_target*, metode yang digunakan kurang lebih sama dengan yang digunakan pada `load_image()`. Lalu program akan melakukan *looping* dari kiri atas gambar, bergerak ke kiri bawah gambar, hal ini adalah *sliding window*. Pada setiap *window*, program akan memanggil cascade yang berhubungan untuk melakukan klasfikasi. Bila *cascade* mengembalikan suatu kelas ikan, maka *looping* untuk area tersebut akan dihentikan dan dilanjutkan ke area berikutnya. Proses

looping ini dijalankan tiga kali untuk area kiri, tengah dan kanan. Estimasi area kiri adalah x = 0 sampai x = 115 - 50, estimasi area tengah adalah x = 116 sampai x = 232 - 50, dan area kanan x = 233 sampai x = 350 -50. Semua loop dikurangi dengan 50, ukuran dari *sub-window*, agar *slidin window* tidak mencoba mengklasifikasi keluar dari areanya atau keluar dari gambar. Terakhir hasil prediksi akan ditentukan oleh suara mayoritas dari ketiga *window*, kecuali kalau ketiga *window* mem-*voting* kelas yang berbeda, maka kelas 0 atau None akan dipilih. Gambar lalu akan dianotasi dengan nama kelas ditulis di kiri atas gambar *output*, kelas dan informasi penting lainnya juga ditambahkan ke nama dokumen gambar *output* agar mudah diperiksa.

4.2.1 Validasi Tes Lapangan

Untuk melakukan validasi lapangan, penulis mengambil gambar ikan secara mandiri ke salah satu petani ikan. Ikan yang akan diklasifikasi adalah spesies ikan Lele, Ikan Nila dan Ikan Emas. Untuk setiap ikan digunakan 9 gambar untuk training, dan dari masing-masing ikan ada satu gambar untuk validasi. Gambar ikan diperlakukan selayaknya training dan validasi sebelumnya, ukuran dirubah menjadi 350 x 200 piksel, dan latar belakang dihilangkan.



Gambar 4.40: Ketiga ikan yang akan diklasifikasi

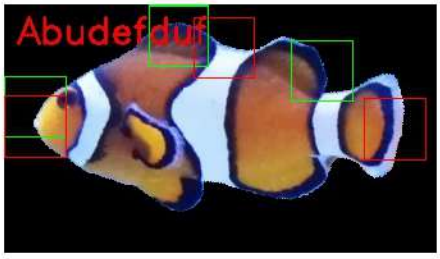
Dari tes lapangan ini lagi-lagi akurasi *classifier* sangatlah rendah. Dari tiga gambar yang diklasifikasi, hanya satu gambar yang berhasil diklasifikasi secara benar

yaitu gambar ikan Emas. Namun hal ini lagi-lagi terjadi karena bias bobot *voting* dari *weak classifier* dan mekanisme *sliding window* yang masih bermasalah. Karena dari ketiga gambar yang diklasifikasi, tiga-tiganya diklasifikasikan sebagai ikan Emas.

4.3 Analisa Hasil

Setelah prediksi menggunakan *sliding window* dilakukan, ditemukan bahwa hanya 12 dari 75 gambar yang berhasil diklasifikasikan dengan benar. Hasil ini jelas bertolak belangan dengan akurasi yang ditunjukan pada tahap *boosting* dimana ketiga *strong classifier* memiliki skor akurasi diatas 70%.

Klasifikasi ikan menggunakan metode *sliding window* menunjukan sebuah problem baru yaitu salahnya prediksi pada lokasi *sub-window* yang tidak seharusnya. *Sliding window* memulai klasifikasi dari pojok kiri atas area klasifikasi dan bergerak kebawah menuju pojok kiri bawah, dengan lokasi akhir klasifikasi pojok kanan bawah. Dalam pergerakannya ini ia akan menolak mayoritas *sub-window* yang tidak memiliki nilai, atau *background*, namun ketika ia menemukan sedikit saja piksel dalam *sub-window*-nya ia akan melakukan klasifikasi.



Gambar 4.41: Gambar tes Amphiprion23. Kotak merah menunjukan ekspektasi klasifikasi yang benar untuk kelas Amphiprion, karena *offset* diletakan disitu. Kotak hijau menunjukan lokasi klasifikasi yang dilakukan *sliding window* ketika mengklasifikasi gambar Amphiprion23

Problem baru lainnya juga keluar pada saat klasifikasi, adalah bias *weak classifier* yang lebih kuat pada awal *cascade* terhadap beberapa kelas tertentu, dalam banyak kasus bias-nya adalah ke kelas ikan Abudefduf. Umumnya *weak classifier* yang paling awal didalam *cascade* memiliki bobot *voting* yang terlalu kuat, sehingga ketika mereka mem-*voting* suatu kelas jumlah total bobot *voting weak classifier* setelahnya tidak dapat mengalahkan *voting weak classifier* paling awal. Dari 50 gambar yang bukan kelas Abudefdu, Abudeduf keluar sebanyak 32 kali dalam klasifikasi.