National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Shivani Chandak
Student ID: x20186762

School of Computing
National College of Ireland

Supervisor: Mr. Taimur Hafeez

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Shivani Chandak
Student ID:	x20186762
Programme:	Data Analytics
Year:	2022
Module:	MSc Research Project
Supervisor:	Mr. Taimur Hafeez
Submission Due Date:	15/08/2022
Project Title:	Configuration Manual
Word Count:	1060
Page Count:	12

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	17th September 2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Shivani Chandak
x20186762

1 Introduction

This Configuration Manual includes all the materials needed to replicate the findings of the research- **"Classification of Severity Levels in Diabetic Retinopathy in Ultra-wide Field Colour Fundus Images using Hybrid Deep Learning Models"**. This document consists of the hardware, software requirements as well as the details of the code written in order to implement the research.

2 System Configuration

2.1 Hardware Specifications

Table 1 demonstrates the hardware specifications of the system on which the research was carried out.

Table 1: Hardware Specifications

RAM	8 GB
Processor	Intel(R) Core(TM) i5-8300H
Speed	2.30 GHz
Operating System	Windows 10, 64 Bit
Storage	1 TB HDD
GPU	NVIDIA GeForce GTX1650

2.2 Software Specifications

2.2.1 Jupiter Notebook from Anaconda Distribution

The Anaconda distribution includes an open-source desktop GUI called Anaconda Navigator. It comprises of Jupiter Notebooks which supports in application of deep learning algorithms as required. Version 6.4.12 consists of all the operations required to construct a deep learning model along-with augmentation and pre-processing features which were needed while implementing this research.

2.2.2 Microsoft Excel

Excel was used to manipulate the CSV.

3 Development of Project

The research was built using the programming language- Python. It was used to implement different sections of the code- data pre-processing, data transformation, model construction and evaluation. The main libraries used were Matplotlib, Pandas, Keras, TensorFlow, Numpy etc.

3.1 Data Collection

The data is available for public use on: **DeepDRid Github**

3.2 Importing Libraries

The required libraries are first imported as shown in Figure 1.

```
# importing relevant libraries
import pandas as pd
import numpy as np
import os
from keras.preprocessing.image import load_img
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder
import zipfile
from keras import layers
from keras import models
from tensorflow.keras.applications import VGG19
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
from sklearn.metrics import confusion_matrix
import tensorflow
from sklearn.svm import SVC
import random
import seaborn as sns
from pylab import rcParams
import cv2
from sklearn import metrics
import tensorflow as tf
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from keras.preprocessing.image import img_to_array
from numpy import expand_dims
import warnings
warnings.filterwarnings('ignore')
```

Figure 1: Importing Libraries

3.3 Data Pre-processing & Transformation

Jupiter notebook is given access to the dataset that was obtained from Github. All the images are contained within a single directory after downloading the dataset. First, a directory structure 'datasetBeforeAugmentation/train' and 'datasetBeforeAugmentation/test' is devised. The code first checks if there is any directory with the name of 'datasetBeforeAugmentation'. If no directory is found, then 'datasetBeforeAugmentation' directory will be made. A similar task is done for the test data as well. Next, the code makes folders from 0 to 4 within both 'train' and 'test' directories if they are not already present. The code grabs images from the common repository and then saves it to the related class in the data directory shown in Figure 2.

```
# making a directory structure to hold the training and testing data into folders 0 - 4

try:
    os.mkdir('datasetBeforeAugmentation')
except:
    print('dataset directory already exists')



# train
try:
    os.mkdir('datasetBeforeAugmentation/Train')
except:
    print('train directory already exists')

for i in range(len(train_info['DR_level'].value_counts())):
    try:
        os.mkdir(f'datasetBeforeAugmentation/train/{i}')
    except:
        print(f'{i} directory already exists')



# test
try:
    os.mkdir('datasetBeforeAugmentation/Test')
except:
    print('test directory already exists')

for i in range(len(train_info['DR_level'].value_counts())):
    try:
        os.mkdir(f'datasetBeforeAugmentation/test/{i}')
    except:
        print(f'{i} directory already exists')
```

Figure 2: Maintaining Directory

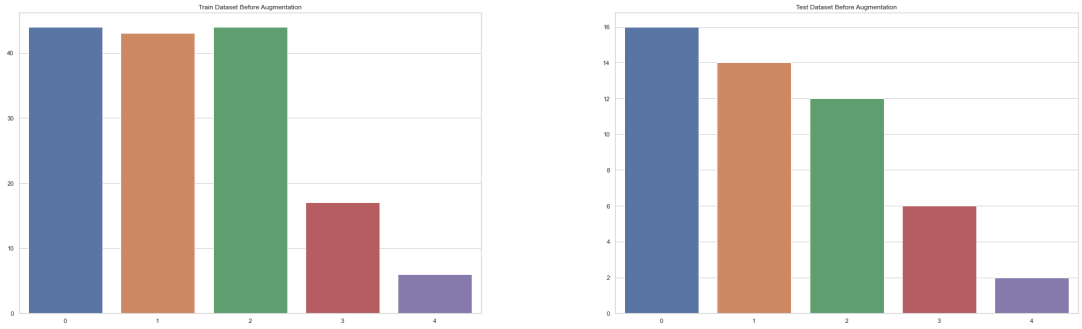

Figure 3: Imbalanced data Before Augmentation

The deep learning model cannot be trained using the original train dataset, as it consists of 155 images divided into 5 classes. Additionally, there was an imbalance in the images for each class as seen in Figure 3. With this few examples in the training data, the model may have a tendency to overfit. To solve this problem, variants are added to the snaps using data augmentation techniques.

Further, the images are augmented and saved into the newly generated directory with a slight shift in zoom, flip and rotation as seen in Figure 4. The image generator is called 50 times for of the class in the train dataset and 35 times for the test dataset. Every time a unique image is generated and saved in the directory. The unbalanced dataset has been made balanced after data augmentation as seen in Figure 5.

3.4 K-Fold Cross Validation

To yield the images in batches to facilitate the training process, a data generator is made. The generator yielded the images in batch size of 1 and sent it to the model for training,

```python
# defining a datagen
datagen = ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    zca_epsilon=0,
    rotation_range=0.1,
    width_shift_range=0,
    height_shift_range=0,
    brightness_range=None,
    shear_range=0.0,
    zoom_range=0.1,
    channel_shift_range=0.0,
    fill_mode='nearest',
    cval=0.0,
    horizontal_flip=True,
    vertical_flip=True,
    rescale=None,
    preprocessing_function=None,
    data_format=None,
    validation_split=0.0,
    dtype=None,
)
```

Figure 4: Data Augmentation

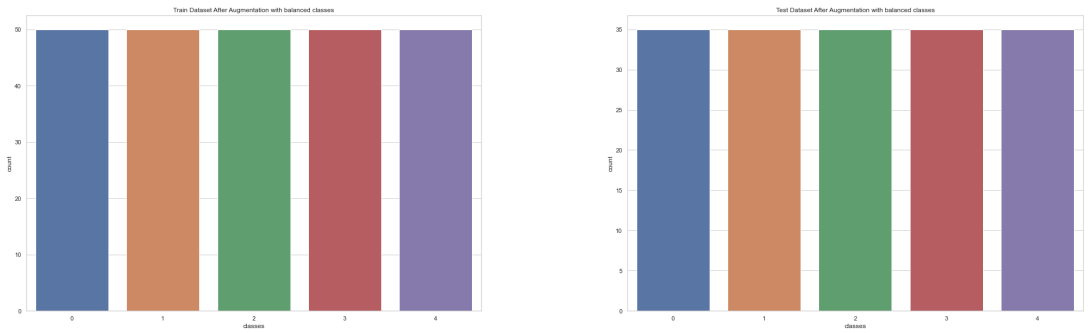

Figure 5: Balanced Data After Augmentation

validation, and testing. Since, the research is using k fold cross validation with 80/20 ratio, for each epoch, 20% of the data is made validation data while the remaining 80% of the data is made train data. The output shape of the data generator was (120, 120, 3) (Figure 6)

```
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    target_size=(120, 120),
                                                    batch_size=1,
                                                    shuffle = True,
                                                    class_mode='categorical',
                                                    subset='training')

# validation data for k fold = 5
validation_generator = train_datagen.flow_from_directory(train_dir,
                                                         target_size=(120, 120),
                                                         batch_size=1,
                                                         shuffle = True, # for k fold
                                                         class_mode='categorical',
                                                         subset='validation')


# test data
test_datagen = ImageDataGenerator(rescale=1./255)

test_generator = test_datagen.flow_from_directory(test_dir,
                                                  target_size=(120, 120),
                                                  batch_size=1,
                                                  shuffle = False,
                                                  class_mode='categorical')
```

Figure 6: K-fold Cross Validation

3.5 Model Implementation

3.5.1 VGG-19

In Figure 8, the research uses base model of VGG-19 with the pre-trained weights of ImageNet dataset to extract the features (low level) out of the images. The input shape from the generator is (120, 120, 3). Then a custom top is added for the classification. The custom top has 4 dense layers. The first layer has 200 neurons followed by the second layer which has 100 neurons. The third layer has 50 neurons and the last layer has 5 neurons (as there are 5 classes). The first three layers had 'relu' as activation function to include some non linarites. Since it is a multi-class problem, the final layer has 'softmax' activation function. "crossentropy" has been used as loss function as this was a classification problem. "Adamax" optimizer has been used since it is among one of the best optimizers for image classification. Additionally, two callbacks are used- one is to save the best weights and other is to monitor the validation loss as seen in Figure 7.

3.5.2 VGG19-SVM

In Figure 9, the research uses base model of VGG-19 with the pre-trained weights of ImageNet dataset to extract the features (low level) out of the images. These low level features are saved in the form of an array. The input shape from the generator is (120, 120, 3). The array is then fed to the Support Vector Machine (SVM) model. The model

```
# function to save best weights only
best_weight = tf.keras.callbacks.ModelCheckpoint(
    filepath='/content/',
    save_weights_only=True,
    monitor='val_accuracy',
    mode='max',
    save_best_only=True)


# function for early stopping
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor="val_accuracy",
    min_delta=0,
    patience=5,
    verbose=2,
    mode="auto",
    baseline=None,
    restore_best_weights=True,
)
```

Figure 7: Function to save Best Weights and monitor Validation Loss

```
# convolution base of Vgg19
np.random.seed(0)
tensorflow.random.set_seed(0)
conv_base = tf.keras.applications.vgg19.VGG19(
    include_top=False,
    weights='imagenet',
    input_tensor=None,
    input_shape=(120,120,3),
    pooling=False,
    classes=5
)

# custom top
model = models.Sequential()
model.add(conv_base)

# Adding our own dense layers
model.add(layers.Flatten())
model.add(layers.Dense(200, activation='relu'))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(50, activation='relu'))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(5, activation='softmax'))


# compiling the model
model.compile(loss='categorical_crossentropy',
              optimizer= 'adamax',
              metrics=['accuracy'])

# training the model
history = model.fit(train_generator,
                    epochs=100,
                    batch_size=25,
                    callbacks=[early_stopping, best_weight],
                    validation_data= validation_generator
                    )
```

Figure 8: VGG-19

SVM has a c parameter equal to 20 to handle the misclassifications. Additionally, the kernel is selected as 'poly'.

```
# convolution base of Vgg19
conv_base = tf.keras.applications.vgg19.VGG19(
    include_top=False,
    weights='imagenet',
    input_tensor=None,
    input_shape=(120,120,3),
    pooling=True,
    classes=5
)


# Make loaded layers as non-trainable. This is important as we want to work with pre-trained weights
for layer in conv_base.layers:
    layer.trainable = False


# Now, let us use features from convolutional network for SVM
feature_extractor=conv_base.predict(train_generator)
features = feature_extractor.reshape(feature_extractor.shape[0], -1)
X_for_SVM = features # This is our X input to SVM


# SVM
SVM_model = SVC(C = 20, kernel = 'poly')


# Train the model on training data
SVM_model.fit(X_for_SVM,train_generator.classes )


# Test
# Send test data through same feature extractor process
X_test_feature = conv_base.predict(test_generator)
X_test_features = X_test_feature.reshape(X_test_feature.shape[0], -1)


# Now predict using the trained SVM model.
prediction_SVM = SVM_model.predict(X_test_features)


# data for perfomence matrices
y_true = test_generator.classes
y_pred = prediction_SVM
enc = OneHotEncoder(handle_unknown='ignore')
enc.fit(y_pred.reshape(-1, 1))
y_p = enc.transform(y_pred.reshape(-1, 1)).toarray()
```

Figure 9: VGG19-SVM

3.5.3 VGG19-RF

In Figure 10, the research uses base model of VGG-19 with the pre-trained weights of ImageNet dataset to extract the features (low level) out of the images. These low level features are saved in the form of an array. The input shape from the generator is (120, 120, 3). The array is then fed to the Random Forest (RF) model. No depth is passed to the model so it can infer the best one.

3.6 Model Evaluation & Results

A generic function is created to analyze all the performance metrices as seen in Figure 11.

```
# convolution base of Vgg19
conv_base = tf.keras.applications.vgg19.VGG19(
    include_top=False,
    weights='imagenet',
    input_tensor=None,
    input_shape=(120,120,3),
    pooling=True,
    classes=5
)


# Make loaded layers as non-trainable. This is important as we want to work with pre-trained weights
for layer in conv_base.layers:
    layer.trainable = False


# Now, let us use features from convolutional network for RF
feature_extractor=conv_base.predict(train_generator)
features = feature_extractor.reshape(feature_extractor.shape[0], -1)
X_for_RF = features # This is our X input to RF


# RF
RF_model = RandomForestClassifier()


# Train the model on training data
RF_model.fit(X_for_RF,train_generator.classes )


# Test
# Send test data through same feature extractor process
X_test_feature = conv_base.predict(test_generator)
X_test_features = X_test_feature.reshape(X_test_feature.shape[0], -1)


# Now predict using the trained RF model.
prediction_RF = RF_model.predict(X_test_features)


# data for perfomence matrices
y_true = test_generator.classes
y_pred = prediction_RF
enc = OneHotEncoder(handle_unknown='ignore')
enc.fit(y_pred.reshape(-1, 1))
y_p = enc.transform(y_pred.reshape(-1, 1)).toarray()
```

Figure 10: VGG19-RF

```python
def performance_evaluation(y_true,y_pred, y_p):

  print('\n\n\n ***PERFORMANCE MATRICES*** \n')


  # precision, recall, fscore
  precision, recall, fscore, none = precision_recall_fscore_support(y_true, y_pred, average='macro')
  print(f'precision = {precision}')
  print(f'recall = {recall}')
  print(f'fscore = {fscore}')


  # accuracy
  print(f'Accuracy: {accuracy_score(y_true, y_pred)}')
  fpr, tpr, thresholds = metrics.roc_curve(y_true, y_pred, pos_label=4)


  # AUC
  print(f'AUC: {metrics.auc(fpr, tpr)}')


  # cohen_kappa_score
  print(f'cohen_kappa_score: {metrics.cohen_kappa_score(y_true, y_pred)}\n')
  # sensitivity and specificity
  res = []
  for l in range(5):
      prec,recall,_,_ = precision_recall_fscore_support(np.array(y_true)==l,
                                                        np.array(y_pred)==l,
                                                        pos_label=True,average=None)
      res.append([l,recall[0],recall[1]])
  print(pd.DataFrame(res,columns = ['class','sensitivity','specificity']))


  # roc curve for multiclasses
  fpr = {}
  tpr = {}
  thresh ={}
  n_classes = y_p.shape[1]
  for i in range(n_classes):
      fpr[i], tpr[i], thresh[i] = roc_curve(y_true, y_p[:,i], pos_label=i)

  plt.figure(figsize = (20,20))
  for i in range(n_classes):
      plt.plot(fpr[i], tpr[i], label=f'Class{i+1}')
  plt.title('Multiclass ROC curve')
  plt.xlabel('False Positive Rate')
  plt.ylabel('True Positive rate')
  plt.legend(loc='upper right', ncol = 10)
  plt.show()


  # confusion matrix
  print('\nConfusion Matrix')
  cnf_matrix= confusion_matrix(y_true, y_pred)
  plt.figure(figsize = (10,7))
  sns.heatmap(cnf_matrix, annot=True, fmt="d")
```

Figure 11: Function for analyzing performance metrices

3.6.1 VGG-19

The confusion matrix and other performance metrics for VGG-19 model can be seen in Figure 12 and Figure 13.

```
precision = 0.8196626984126985
recall = 0.8
fscore = 0.7938549342186088
Accuracy: 0.8
AUC: 0.7956122448979592
cohen_kappa_score: 0.75

   class  sensitivity  specificity
0      0     0.921429     0.971429
1      1     0.921429     0.600000
2      2     0.914286     0.857143
3      3     0.992857     1.000000
4      4     1.000000     0.571429
```

Figure 12: Performance Metrices: VGG-19

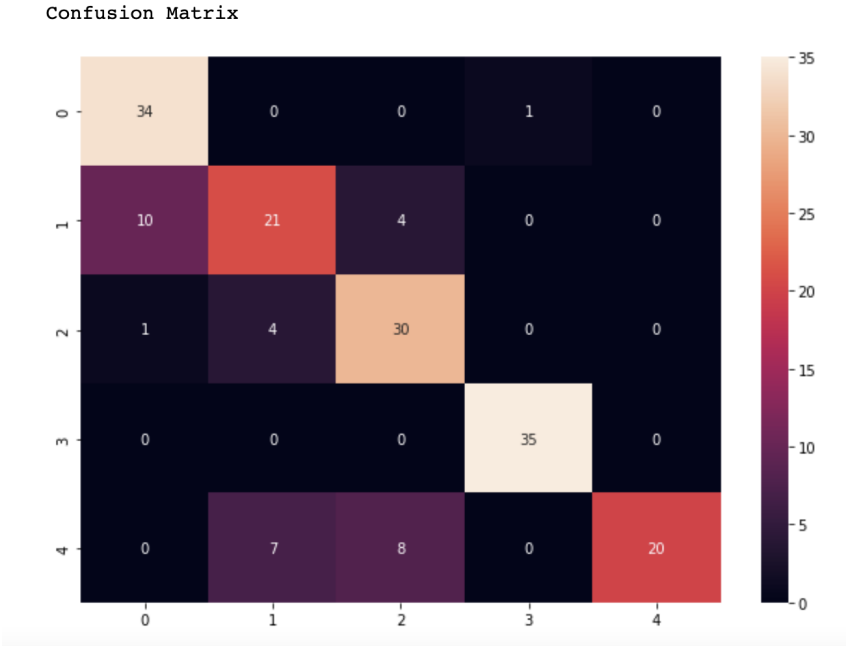

Figure 13: Confusion Matrix: VGG-19

3.6.2 VGG19-SVM

The confusion matrix and other performance metrics for VGG19-SVM model can be seen in Figure 14 and Figure 15.

3.6.3 VGG19-RF

The confusion matrix and other performance metrics for VGG19-RF model can be seen in Figure 16 and Figure 17.

```
precision = 0.44850515816082537
recall = 0.4171428571428571
fscore = 0.41369699302217927
Accuracy: 0.41714285714285715
AUC: 0.6803061224489795
cohen_kappa_score: 0.27142857142857146

   class  sensitivity  specificity
0      0     0.664286     0.371429
1      1     0.950000     0.142857
2      2     0.892857     0.371429
3      3     0.928571     0.685714
4      4     0.835714     0.514286
```

Figure 14: Performance Metrices: VGG19-SVM

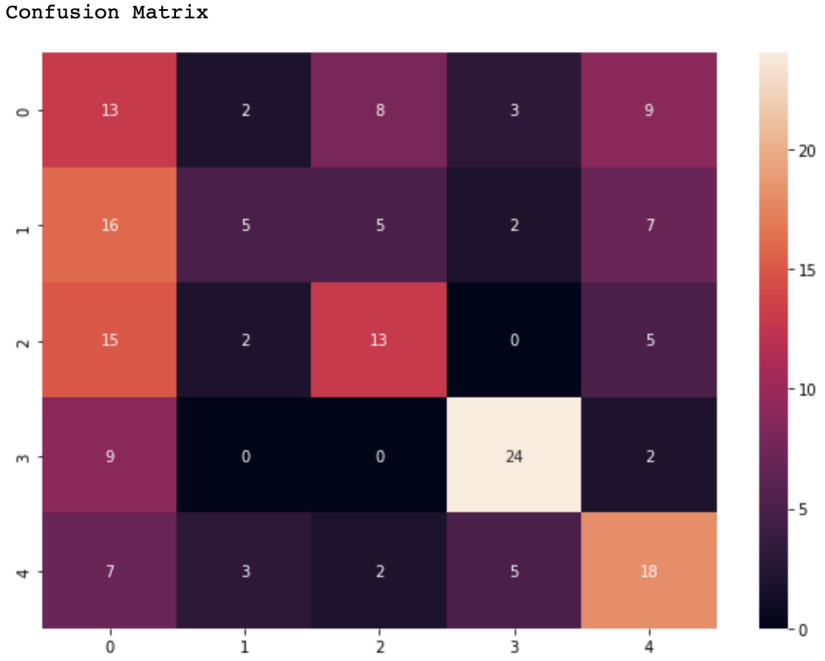

Figure 15: Confusion Matrix: VGG19-SVM

```
precision = 0.5396913231695841
recall = 0.56
fscore = 0.5373434892785534
Accuracy: 0.56
AUC: 0.766734693877551
cohen_kappa_score: 0.44999999999999996

   class  sensitivity  specificity
0      0     0.835714     0.485714
1      1     0.907143     0.314286
2      2     0.921429     0.285714
3      3     0.935714     1.000000
4      4     0.850000     0.714286
```

Figure 16: Performance Metrices: VGG19-RF

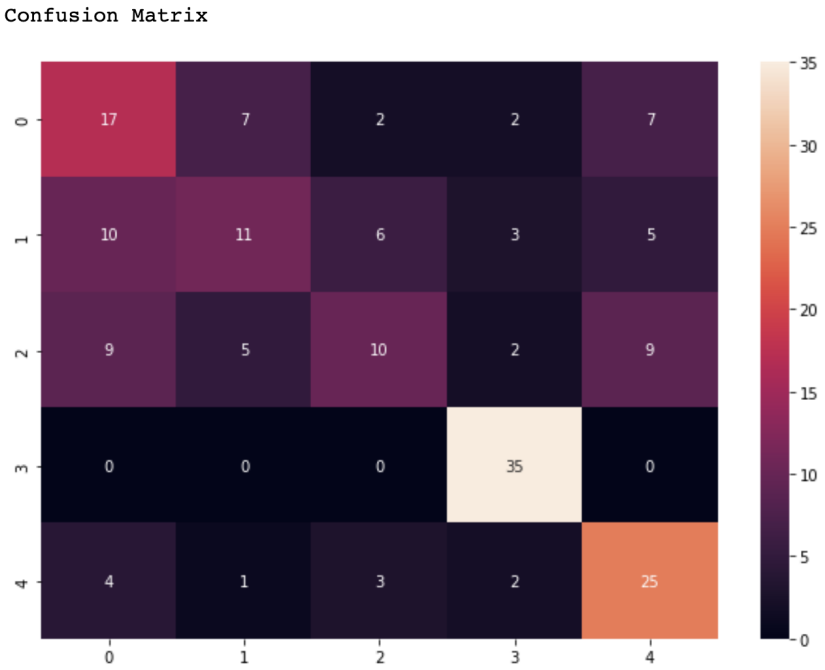

Figure 17: Confusion Matrix: VGG19-RF