National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Maria Migrova
Student ID: x21146021

School of Computing
National College of Ireland

Supervisor:	Jorge Basilio
Industry Supervisor:	Mihai Ilie

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Maria Migrova
Student ID:	x21146021
Programme:	Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Jorge Basilio
Submission Due Date:	14/08/2023
Project Title:	Configuration Manual
Word Count:	713
Page Count:	10

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:	Maria Migrova
Date:	18th September 2023

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

Maria Migrova
x21146021

14/08/2023

1 Introduction

The Configuration Manual encompasses software and hardware configurations, libraries, and key code excerpts pertinent to each step of the implementation process. Its purpose is to guide the replication of the research project titled "Optimizing Data Collection Process in the Autonomous Driving Industry Using Machine Learning." This documentation is designed to facilitate the precise reproduction of the project's procedures, ensuring accuracy and enabling further exploration of the study's outcomes.

2 System Requirements

2.1 Hardware configuration

Table 1 presents the Hardware configurations, outlining the necessary components for the project's execution.

Table 1: Hardware Specification

Processor	AMD Ryzen 7 2700X Eight-Core Processor
Clock Speed	3.70 GHz
RAM	32.0 GB

2.2 Software configuration

Table 2 presents the Software configurations.

Table 2: Software Specification

Software Type	Software Name	Version
Programming Language	Python	3.8.5
Notebook	Jupyter Notebook	6.1.4
Platform	Tensorflow	2.6.0

2.3 Libraries and Packages

Table 3 describes the libraries and packages used in this project and their usage.

Table 3: Libraries Used and Descriptions

Library	Version	Description
os	N/A	Provides functions to interact with the operating system, used for file and directory operations.
cv2	4.5.3	OpenCV library for computer vision tasks such as image processing and analysis.
pandas	1.3.3	Data manipulation and analysis library, used for handling tabular data.
numpy	1.21.2	Numerical computing library, used for mathematical operations on arrays and matrices.
sklearn.model_selection	0.24.2	Part of scikit-learn library, used for model selection and evaluation.
tensorflow.keras.preprocessing.image	2.6.0	Part of TensorFlow library, used for image data preprocessing.
tensorflow.keras.models	2.6.0	Part of TensorFlow library, used for building and training models.
tensorflow.keras.layers	2.6.0	Part of TensorFlow library, provides layers for building neural networks.
sklearn.preprocessing	0.24.2	Part of scikit-learn library, used for data preprocessing and scaling.
tensorflow.keras.callbacks	2.6.0	Part of TensorFlow library, used for defining custom callbacks during training.
tensorflow.keras.optimizers	2.6.0	Part of TensorFlow library, provides optimization algorithms for neural networks.
matplotlib.pyplot	3.4.3	Used for creating visualizations such as plots and graphs.
collections	N/A	Provides specialized data structures, used for managing and manipulating collections of items.
skimage	0.18.3	Part of sci-kit-image library, used for image processing and computer vision tasks.
tabulate	0.8.9	Used for formatting tabular data, such as displaying results in a table format.
json	N/A	Used for working with JSON data format.
openpyxl	3.0.7	Library for working with Excel files (xlsx), used for data manipulation.
glob	N/A	Provides file path pattern matching, used for file operations.
re	N/A	Regular expression library, used for pattern matching and manipulation of strings.

3 Research Implementation

3.1 Data Preprocessing

(Figure 1) Represents the function specified for metadata extraction from multiple preannotation.json files.

```
def extract_info_from_json(json_data, json_file_name):
    data = json.loads(json_data)
    algorithm = None
    scene_environment = None
    scene_sky_cover = None
    scene_climatic_conditions = None
    scene_surface_conditions = None
    scene_lighting = None
    scene_surface_type = None
    scene_country = None
    soiling_type = None

    if data.get("type") == "response_video_data" and data.get("content"):
        content = data["content"]

        # Check if label_structures section exists
        label_structures = content.get("label_structures", [])
        for structure in label_structures:
            algorithm = structure.get("algorithm")
            break

        # Check if scene_attributes section exists
        scene_attributes = content.get("scene_attributes", [])
        if scene_attributes:
            for attr in scene_attributes:
                if "SAWDL" in attr.get("algorithms", []):
                    values = attr.get("values", [])
                    if values:
                        value = values[0].get("name")
                        if attr["name"] == "scene_environment":
                            scene_environment = value
                        elif attr["name"] == "scene_sky_cover":
                            scene_sky_cover = value
                        elif attr["name"] == "scene_climatic_conditions":
                            scene_climatic_conditions = value
                        elif attr["name"] == "scene_surface_conditions":
                            scene_surface_conditions = value
                        elif attr["name"] == "scene_lighting":
                            scene_lighting = value
                        elif attr["name"] == "scene_surface_type":
                            scene_surface_type = value
                        elif attr["name"] == "scene_country":
                            scene_country = value
```

Figure 1: Metadata Extract function to excel

(Figure 2) shows the function to resize images in the specific folder, this one resizes all the images to 240 x 240 format using os and cv2 library.

```python
import os
import cv2

# Function to resize images in the specified folder and its subfolders
def resize_images(main_folder, output_folder, target_size=(240, 240)):
    # Creating the output folder if it doesn't exist
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # Creating a log file to keep track of skipped files
    log_file = os.path.join(output_folder, 'skipped_files.txt')
    with open(log_file, 'w') as log:
        log.write("Files skipped due to errors:\n")

    # Going through all subfolders and find image files
    for root, dirs, files in os.walk(main_folder):
        for image_file in files:
            # Checking if the file is an image
            if image_file.lower().endswith(('.png', '.jpg', '.jpeg')):
                # Reading the image using OpenCV
                image_path = os.path.join(root, image_file)
                img = cv2.imread(image_path)

                # Checking if the image is valid
                if img is None:
                    with open(log_file, 'a') as log:
                        log.write(image_path + '\n')
                    print(f"Skipping {image_file} due to read error.")
                    continue

                # Resizing the image to the target size
                resized_img = cv2.resize(img, target_size, interpolation=cv2.INTER_AREA)

                # Getting the relative path of the image to the main folder
                relative_path = os.path.relpath(root, main_folder)

                # Creating subdirectories in the output folder if they don't exist
                output_subfolder = os.path.join(output_folder, relative_path)
                if not os.path.exists(output_subfolder):
                    os.makedirs(output_subfolder)

                # Saving the resized image to the output folder
                output_path = os.path.join(output_subfolder, image_file)
                cv2.imwrite(output_path, resized_img)

                print(f"Image {image_file} resized and saved to {output_path}")

# Asking the user for the path to the main folder containing subfolders with images to be resized
main_folder = input("Enter the path to the main folder: ")

# Asking the user for the path to the output folder to save the resized images
output_folder = input("Enter the path to the output folder: ")

# Resizing the images in the main folder and its subfolders and save them to the output folder
resize_images(main_folder, output_folder)
```

Figure 2: Image resizing

(Figure 3) represents the augmentation script that was used to adjust contrast and brightness on the images to create new augmented images.

```python
import os
import cv2
import random
import pandas as pd
from skimage import exposure

# Function to augment images in the specified folder and its subfolders
def augment_data(main_folder, output_folder, augment_factor):
    # Creating the output folder if it doesn't exist
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # Loading the CSV file containing image labels
    labels_df = pd.read_csv(r"C:\Users\mmigrova\Desktop\Masters\Code\Dataset\image_metadata_mapping.csv")

    # Creating a dictionary to store the mapping between image names and their labels
    image_label_mapping = {}

    # Going through all subfolders and find image files
    for root, dirs, files in os.walk(main_folder):
        for image_file in files:
            # Checking if the file is an image
            if image_file.lower().endswith(('.png', '.jpg', '.jpeg')):
                # Getting the image name without the extension
                image_name = os.path.splitext(image_file)[0]

                # Getting the corresponding label from the DataFrame
                label_row = labels_df.loc[labels_df['Name'] == image_name]
                if not label_row.empty:
                    # Getting the labels from the row (assuming columns 'scene_sky_cover', 'scene_climatic_conditions', etc.)
                    label_sky_cover = label_row['scene_sky_cover'].values[0]
                    label_climatic_conditions = label_row['scene_climatic_conditions'].values[0]
                    label_surface_conditions = label_row['scene_surface_conditions'].values[0]
                    label_lighting_conditions = label_row['lighting_conditions'].values[0]

                    # Saving the labels in the image_label_mapping dictionary
                    image_label_mapping[image_name] = {
                        'scene_sky_cover': label_sky_cover,
                        'scene_climatic_conditions': label_climatic_conditions,
                        'scene_surface_conditions': label_surface_conditions,
                        'lighting_conditions': label_lighting_conditions
                    }

                # Loading the image using OpenCV
                image_path = os.path.join(root, image_file)
                img = cv2.imread(image_path)

                # Performing data augmentation 'augment_factor' times
                for i in range(augment_factor):
                    # Randomly selecting an augmentation technique
                    augmentation_technique = random.choice(["brightness", "contrast"])

                    # Applying the selected augmentation technique
                    if augmentation_technique == "brightness":
                        # Randomly adjusting the brightness of the image
                        alpha = random.uniform(0.8, 1.2)
                        augmented_img = exposure.adjust_gamma(img, gamma=alpha)
                    elif augmentation_technique == "contrast":
                        # Randomly adjusting the contrast of the image
                        beta = random.uniform(0.8, 1.2)
                        augmented_img = cv2.convertScaleAbs(img, alpha=beta, beta=0)

                    # Saveing the augmented image in the output folder with a new name
                    output_image_name = f"{image_name}_augmented_{i}.png"
```

Figure 3: Image Augmentation

4 Data Splitting

Here we can see the stratified sampling used in the project for all the models to split the dataset based on all the classes.

```python
# Filtering out invalid image filenames
metadata_df['Path'] = metadata_df['Name'].apply(lambda x: os.path.join(base_image_directory, x))
metadata_df = metadata_df[metadata_df['Path'].apply(lambda x: os.path.exists(x))]


# Create a combined label based on all four classes
metadata_df['combined_label'] = metadata_df['scene_sky_cover'].astype(str) + '_' + \
                                metadata_df['scene_climatic_conditions'].astype(str) + '_' + \
                                metadata_df['scene_surface_conditions'].astype(str) + '_' + \
                                metadata_df['lighting_conditions'].astype(str)

# Defining the target column
target_column = 'combined_label'

# Creating label encoder
label_encoder = LabelEncoder()

# Encoding the target column
metadata_df[target_column] = label_encoder.fit_transform(metadata_df[target_column])

# Splitting the dataset into 80% training, 10% validation, and 10% test
train_df, temp_df = train_test_split(metadata_df, test_size=0.2, stratify=metadata_df[target_column], random_state=42)
validation_df, test1_df = train_test_split(temp_df, test_size=0.5, stratify=temp_df[target_column], random_state=42)

# Printing the sizes of each set
print("Train set size:", len(train_df))
print("Validation set size:", len(validation_df))
print("Test set size:", len(test1_df))

# Saving the dataframes into separate sheets of an Excel file
excel_file_path = 'E:/Valeo Woodspace data/FINAL_DATASET/split_data1.xlsx'
with pd.ExcelWriter(excel_file_path) as writer:
    train_df.to_excel(writer, sheet_name='train', index=False)
    validation_df.to_excel(writer, sheet_name='validation', index=False)
    test1_df.to_excel(writer, sheet_name='test', index=False)

# Printing the mapping of labels to numerical representations
label_mapping = label_encoder.classes_
print("Label Mapping:")
print("Label Name\t| Numerical Label")
print("-" * 30)
for i, label_name in enumerate(label_mapping):
    print(f"{label_name}\t| {i}")

# Counting the combinations
combination_counts = metadata_df['combined_label'].value_counts()
print("\nCombination Counts:")
print(combination_counts)

# Saving the dataframes into separate sheets of an Excel file
excel_file_path = 'E:/Valeo Woodspace data/FINAL_DATASET/split_data.xlsx'
with pd.ExcelWriter(excel_file_path) as writer:
    train_df.to_excel(writer, sheet_name='train', index=False)
    validation_df.to_excel(writer, sheet_name='validation', index=False)
    test1_df.to_excel(writer, sheet_name='test', index=False)
```

Figure 4: Stratified Sampling and dataset split

5 Final Preprocessing for the models

(Figure 5) This code sets up image data generators for training and validation data. It rescales pixel values, defines the batch size, and generates batches of images from data frames, preparing them for a model. The target column is specified as 'combined_label' for classification tasks.

```python
# Creating an ImageDataGenerator
datagen = ImageDataGenerator(rescale=1./255)

# Creating the data generators
#Specifying the batch size
batch_size = 128

#instance of ImageDataGenerator that generates batches of training data for our model during training.
train_generator = datagen.flow_from_dataframe(
    dataframe=train_df,
    x_col='Path',
    y_col=target_column,
    target_size=(240, 240),
    batch_size=batch_size,
    class_mode='raw',
    subset='training'
)

#instance of ImageDataGenerator that generates batches of validation data for evaluating our model during training.
validation_generator = datagen.flow_from_dataframe(
    dataframe=validation_df,
    x_col='Path',
    y_col=target_column,
    target_size=(240, 240),
    batch_size=batch_size,
    class_mode='raw'
)

# Defining the target column
target_column = 'combined_label'
```

Figure 5: Final preprocessing of the data

6 Model creating

6.1 EfficientNetB1 Model

Creating the model

[14]:
```python
# Creating the final model with the output layer for scene_sky_cover
num_classes_output = 3
base_model = EfficientNetB1(include_top=False, weights='imagenet', input_shape=(240, 240, 3))
x = layers.GlobalAveragePooling2D()(base_model.output)
x = Dropout(0.5)(x)  # Adding dropout layer with dropout rate of 0.5

# Adding a dense layer
x = layers.Dense(512, activation='relu')(x)
x = layers.Dropout(0.9)(x)  # Add dropout here

output = layers.Dense(num_classes_output, activation='softmax', name='output')(x)
model = Model(inputs=base_model.input, outputs=output)
```

[15]:
```python
# Compiling the model with appropriate loss function for scene_sky_cover
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',  # Use 'sparse_categorical_crossentropy' for integer labels
    metrics=['accuracy']
)
```

[16]:
```python
# Defining the ModelCheckpoint callback
model_checkpoint = ModelCheckpoint(
    filepath='E:/Models/EfficientNet/model_checkpoint_{epoch:02d}.h5',  # Save the model with the epoch number
    monitor='val_accuracy',  # Monitor validation accuracy
    save_best_only=True,  # Save only the best model
    save_weights_only=False,  # Save the entire model, not just weights
    mode='max',  # Maximize the validation accuracy
    verbose=1  # Print messages when saving the model
)
```

Figure 6: EfficientNetB1

This code creates a classification model using EfficientNetB1 architecture for scene sky-cover prediction. It adds layers for global average pooling, dropout, dense, and output. The model is compiled with 'adam' optimizer and 'sparse_categorical_crossentropy' loss. A ModelCheckpoint callback is defined to save the best model based on validation accuracy. (Figure 6)

6.2 Simple CNN Model

```
#Number of classes in our classification problem
num_classes_output = 8

#Sequential model
model = Sequential()

# Adding a Convolutional layer with 32 filters, each of size (3, 3) using ReLU activation
# Input shape is (240, 240, 3), meaning images are 240x240 pixels with 3 color channels (RGB)
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(240, 240, 3)))
model.add(MaxPooling2D((2, 2)))

# Adding a MaxPooling layer to downsample the image representation by taking the maximum value in each 2x2 region
model.add(MaxPooling2D((2, 2)))

# Adding another Convolutional layer with 64 filters and ReLU activation
model.add(Conv2D(64, (3, 3), activation='relu'))

# Adding another MaxPooling layer
model.add(MaxPooling2D((2, 2)))

# Flattening the 2D image representation into a 1D vector
model.add(Flatten())

# Adding a Dense (fully connected) layer with 128 units and ReLU activation
model.add(Dense(128, activation='relu'))

# Adding a Dropout layer to randomly deactivate 50% of the neurons to prevent overfitting
model.add(Dropout(0.5))

# Adding the final Dense layer with the number of units equal to the number of classes
# Using softmax activation for multi-class classification
model.add(Dense(num_classes_output, activation='softmax'))

# Compiling the model with the Adam optimizer and sparse categorical cross-entropy loss
# Using accuracy as a metric to monitor during training
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Defining the EarlyStopping callback
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5, restore_best_weights=True)

# Defining the ModelCheckpoint callback
model_checkpoint = ModelCheckpoint(
    filepath='E:/Models/Final_model/model_checkpoint_{epoch:02d}.h5',  # Save the model with the epoch number
    monitor='val_accuracy',  # Monitor validation accuracy
    save_best_only=True,  # Save only the best model
    save_weights_only=False,  # Save the entire model, not just weights
    mode='max',  # Maximize the validation accuracy
    verbose=1  # Print messages when saving the model
)
```

Figure 7: CNN

This code defines a sequential convolutional neural network (CNN) model for image classification. The model consists of convolutional and max-pooling layers, followed by dense layers and dropout regularization. It's compiled with the Adam optimizer and sparse categorical cross-entropy loss. EarlyStopping and ModelCheckpoint callbacks are used for monitoring and saving the best model during training. (Figure 7)

7 Model Training

```
# Training the model with early stopping
history = model.fit(
    train_generator,
    steps_per_epoch=len(train_generator),
    validation_data=validation_generator,
    validation_steps=len(validation_generator),
    epochs=100,
    callbacks=[early_stopping,model_checkpoint]
)
```

Figure 8: Model Training

(Figure 8) code trains the previously defined CNN model using the provided data generators for training and validation. It specifies the number of training and validation steps based on the length of the generators. The training process occurs over a maximum of 100 epochs. The EarlyStopping callback is utilized to prevent overfitting by monitoring the validation accuracy and stopping training if it does not improve for a certain number of consecutive epochs. The ModelCheckpoint callback saves the best model based on validation accuracy during training.

8 Model Testing

```python
# Using the trained model to predict on the test data
predictions = model.predict(test_generator)

# Converting predicted probabilities to class labels
predicted_labels = np.argmax(predictions, axis=1)

# Getting the true labels from the test generator
true_labels = test1_df[target_column].values

# Getting the filenames of the test images
test_filenames = test1_df['Path'].values

# Evaluating the model on the test set
test_loss, test_accuracy = model.evaluate(test_generator)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)

# Printing the predictions along with true labels and filenames
for i in range(len(test_filenames)):
    filename = os.path.basename(test_filenames[i])
    true_label = true_labels[i]
    predicted_label = predicted_labels[i]

    print(f"Filename: {filename}, True Label: {true_label}, Predicted Label: {predicted_label}")

# Create a dictionary to store the count of wrongly predicted labels
wrongly_predicted_counts = Counter()

for i in range(len(test_filenames)):
    true_label = true_labels[i]  # Use true_labels instead of true_labels_encoded
    predicted_label = predicted_labels[i]

    if true_label != predicted_label:
        wrongly_predicted_counts[true_label] += 1

# Extract label names and their corresponding counts
labels = list(wrongly_predicted_counts.keys())
counts = list(wrongly_predicted_counts.values())
```

Figure 9: Model Testing

This code applies the trained model to predict class labels on the test data using the model.predict method. The predicted probabilities are then converted to class labels by selecting the indices with the highest probabilities. The true class labels from the test set are retrieved, along with the filenames of the test images. The model's performance on the test set is evaluated using the evaluate method, which calculates the test loss and accuracy. The predictions, along with true and predicted labels, are printed for each test image. Additionally, the code analyzes wrongly predicted labels by creating a dictionary that stores the count of wrongly predicted labels. Finally, the labels and their corresponding counts are extracted to analyze which classes were more frequently misclassified. (Figure 9)