National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Nancy Saini
Student ID: x22236040

School of Computing
National College of Ireland

Supervisor: Vikas Tomer

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Nancy Saini
Student ID:	x22236040
Programme:	Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Vikas Tomer
Submission Due Date:	12/08/2024
Project Title:	Configuration Manual
Word Count:	1370
Page Count:	9

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:	Nancy Saini
Date:	12th August 2024

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

Nancy Saini
x22236040

1 Introduction

This configuration manual details the steps necessary to set up and deploy an image detection system that was developed for the MSc Research Project. The system utilizes a multimodal approach, incorporating Histogram of Oriented Gradients (HOG), Local Binary Patterns (LBP), and Convolutional Neural Networks (CNN) to extract features from images. These features are then used by machine learning models, specifically Support Vector Machines (SVM) and Logistic Regression, to distinguish between real and AI-generated images. This system is intended for use in applications such as digital forensics and content verification, providing users with a reliable method for assessing image authenticity. By following this manual, users can replicate the research setup, train models, and conduct evaluations to understand the system's capabilities and performance.

2 System Requirements

This section has details about the hardware and software requirements for the AI-generated image detection system, ensuring optimal performance and reliability.

Hardware Requirements: The system should have an Intel Core i7 processor or equivalent, with a minimum of 16 GB RAM to handle computational tasks efficiently. At least 100 GB of free disk space is advised for storing datasets and model files. For smooth deep learning operations, an NVIDIA GPU with CUDA support is advised.

Software Requirements: The system should run Windows 10/11 or on a recent Linux version, such as Ubuntu 20.04. Python 3.8 or higher is required, using Jupyter Notebook or PyCharm as the development environment.

important Python libraries include TensorFlow and Keras for neural network development, OpenCV for image processing, and scikit-learn for machine learning tasks. NumPy and Pandas support numerical and data operations, while Matplotlib and Seaborn are used for visualization. The imgaug library handles data augmentation, and joblib is used for parallel computing and model serialization.

Additional Tools: For those utilizing Google Colaboratory, ensure account access to this cloud-based platform with GPU support. If using a local NVIDIA GPU, install the CUDA toolkit and cuDNN library for GPU acceleration.

3 Installation and Setup

This section outlines the steps to configure the environment for running the AI-generated image detection system, based on the dataset downloaded locally.

3.1 Python Environment Setup

1. **Install Python:** Download and install Python from `https://www.python.org/`.
2. **Create a Virtual Environment:**

```
python -m venv ai_image_detection_env
```

3. **Activate the Environment:**
   - **Windows:** `.\ai_image_detection_env\Scripts\activate`
   - **Linux/Mac:** `source ai_image_detection_env/bin/activate`

3.2 Install Required Libraries

Run the following command to install necessary libraries:

```
pip install tensorflow keras opencv-python scikit-learn numpy pandas matplotlib seabo
```

3.3 Dataset Preparation

1. **Download Dataset:** The dataset was downloaded from `https://www.kaggle.com/datasets/xhlulu/140k-real-and-fake-faces/data`.
2. **Local Organization:** Organize the dataset locally into `TRAIN`, `TEST`, and `VALID` directories, placing these folders in the same directory as your Jupyter Notebook.

Figure 1: Local directory structure for dataset organization.

3.4 Optional Setup for GPU Acceleration

- **Google Colaboratory:** Access Google Colab for GPU support if needed.
- **CUDA and cuDNN:** Install for enhanced performance on local NVIDIA GPUs.

This setup ensures a consistent environment for executing and testing the image detection models effectively.

4 Data Preprocessing

Data preprocessing is a crucial step in preparing the dataset for analysis and modeling. This section describes the techniques used to preprocess the images for the AI-generated image detection system.

4.1 Importing Libraries

To run the data preprocessing steps, many libraries are imported, including OpenCV for image handling, NumPy for numerical operations, and others for additional processing and analysis. These libraries provide the necessary functions and tools for efficient data manipulation.

```
# Importing the required libraries
import os
import cv2
import numpy as np
import pandas as pd
import glob
import matplotlib.pyplot as plt
import seaborn as sns
import imgaug.augmenters as iaa
from tqdm import tqdm
from skimage.feature import hog, local_binary_pattern
from sklearn.preprocessing import StandardScaler
from joblib import Parallel, delayed
import inspect
from datetime import datetime
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.image import img_to_array, load_img
```

Figure 2: Screenshot of Imported Libraries in the Jupyter Notebook.

4.2 Image Preprocessing Steps

1. **Image Loading:** Images are loaded using OpenCV to read and process them from the local directories.

2. **Resizing:** Images are resized to 128×128 pixels to ensure uniformity across the dataset.

3. **Grayscale Conversion:** Images are converted to grayscale to simplify the feature extraction process and reduce computational complexity.

4. **Normalization:** Pixel values are scaled to the range [0, 1] by dividing by 255 to standardize the input data for the models.

```python
#preprocess an image
def preprocess_image(img):
    if img is not None:
        img = cv2.resize(img, image_size)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        img = cv2.GaussianBlur(img, (3, 3), 0)
        img = img / 255.0
    return img

```

Figure 3: Image Preprocessing Pipeline: Loading, Resizing, Grayscale Conversion, and Normalization.

4.3 Stratified Sampling

- **Balanced Dataset Creation:** Stratified sampling is used to create balanced datasets for training, validation, and testing, ensuring equal representation of real and fake images in each dataset split.
- **Sample Sizes:**
  - `TRAIN`: 10,000 samples (5,000 real, 5,000 fake)
  - `TEST`: 2,500 samples (1,250 real, 1,250 fake)
  - `VALID`: 2,500 samples (1,250 real, 1,250 fake)

```python
#stratify and sample data to ensure both classes are included
def stratified_sample(df, n_samples, seed=42):
    df_real = df[df['label'] == 1]
    df_fake = df[df['label'] == 0]
    n_samples_per_class = n_samples // 2
    df_real_sampled = df_real.sample(n=n_samples_per_class, random_state=seed)
    df_fake_sampled = df_fake.sample(n=n_samples_per_class, random_state=seed)
    return pd.concat([df_real_sampled, df_fake_sampled]).reset_index(drop=True)

```

Figure 4: Stratified Sampling Process

4.4 Image Augmentation (Optional)

- **Flipping and Cropping:** Random horizontal flips and cropping are applied to increase dataset diversity and improve model generalization.
- **Contrast Adjustment:** Image contrast is modified to enhance feature visibility.

```python
#apply augmentation #Horizontal flip with a 50% probability
def augment_image(img):
    seq = iaa.Sequential([iaa.Fliplr(0.5), iaa.Crop(percent=(0, 0.1)), iaa.LinearContrast((0.75, 1.5))])   #Horizontal flip
    img_aug = seq(image=(img * 255).astype(np.uint8)) / 255.0
    return img_aug

```

Figure 5: Image Augmentation Techniques

This preprocessing pipeline prepares the data for feature extraction and subsequent modeling steps, ensuring that the input data is both consistent and informative.

5 Feature Extraction

Feature extraction is an essential process in transforming raw data into a format suitable for machine learning models. In this section, we describe the methods used to extract meaningful features from the images.

5.1 HOG (Histogram of Oriented Gradients)

- **Overview:** HOG features capture the structure and edges of images by computing gradients and orientation histograms.
- **Extraction Process:** Each image is divided into small connected regions, and for each region, a histogram of gradient directions is calculated.
- **Purpose:** HOG is particularly useful for object detection and identifying spatial patterns in the images.

```
# to extract HOG features from an image
def extract_hog_features(img):
    if img is not None:
        features, _ = hog(img, pixels_per_cell=(8, 8), cells_per_block=(2, 2), visualize=True)
        return features
    return None
```

Figure 6: HOG Feature Extraction Process

5.2 LBP (Local Binary Patterns)

- **Overview:** LBP is a simple yet powerful texture descriptor, used to summarize local structures in images.
- **Extraction Process:** For each pixel in a grayscale image, compare it with its neighbors and encode the result as a binary number.
- **Purpose:** LBP effectively captures local texture features, making it valuable for image classification tasks.

```

# extract LBP features from an image
def extract_lbp_features(img):
    if img is not None:
        radius = 3
        n_points = 8 * radius
        lbp = local_binary_pattern(img, n_points, radius, method='uniform')
        n_bins = int(lbp.max() + 1)
        lbp_hist, _ = np.histogram(lbp, density=True, bins=n_bins, range=(0, n_bins))
        return lbp_hist
    return None

```

Figure 7: LBP Feature Extraction Process

5.3 CNN Features using VGG16

- **Overview:** VGG16, a pre-trained Convolutional Neural Network (CNN), is utilized to extract high-level features from images.
- **Extraction Process:** Images are resized to 224×224 pixels and passed through the VGG16 model (without the top classification layers) to obtain feature maps.
- **Purpose:** These features capture complex patterns and hierarchies in images, enabling robust image classification.

```
25
26 # extract CNN features using VGG16
27 def extract_cnn_features(img_path, model, verbose=False):
28     if verbose:
29         print(f"Extracting CNN features for {img_path}")
30     img = load_img(img_path, target_size=image_size)
31     img_array = img_to_array(img)
32     img_array = np.expand_dims(img_array, axis=0)
33     img_array /= 255.0
34     features = model.predict(img_array, verbose=0)
35     return features.flatten()
36
37 # Initializing VGG16 model outside the parallel function
38 def initialize_model():
39     base_model = VGG16(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
40     model = Model(inputs=base_model.input, outputs=base_model.output)
41     return model
42
43 model = initialize_model()
44
```

Figure 8: CNN Feature Extraction using VGG16

5.4 Combined Feature Vector

- **Integration:** Features extracted from HOG, LBP, and VGG16 are concatenated to form a comprehensive feature vector for each image.
- **Advantage:** This combination leverages the strengths of both traditional and deep learning-based feature extraction techniques, improving model performance.

```python

# Extracting combined features (HOG, LBP, and CNN)
def extract_combined_features_debug(img_path, verbose=False):
    try:
        if verbose:
            print(f"Processing image: {img_path}")
        img = load_image(img_path)
        if img is None:
            return None
        img_gray = preprocess_image(img)
        hog_features = extract_hog_features(img_gray)
        lbp_features = extract_lbp_features(img_gray)
        cnn_features = extract_cnn_features(img_path, model, verbose=False)
        combined_features = np.concatenate((hog_features, lbp_features, cnn_features))
        return combined_features
    except Exception as e:
        if verbose:
            print(f"Error extracting features from {img_path}: {e}")
        return None


```

Figure 9: Combined Feature Vector Creation

This feature extraction pipeline transforms raw image data into a rich and informative format, facilitating effective model training and analysis.

6 Model Training and Evaluation

The model training and evaluation process involves training machine learning models using the extracted features and assessing their performance on the test dataset. This section provides a detailed description of the models and evaluation techniques used.

6.1 Support Vector Machine (SVM) Training

- **Overview:** Support Vector Machines are used to classify the images based on the extracted features.
- **Training Process:**
  - The SVM model is trained using the combined feature vectors derived from HOG, LBP, and CNN features.
  - The hyperparameters for the SVM model, such as kernel type and regularization parameter C, are optimized using grid search cross-validation.
- **Purpose:** SVMs are effective for binary classification tasks and can handle high-dimensional feature spaces.

```python
# Creating a smaller subset for debugging
df_train_debug = stratified_sample_ensure_labels(df_train_sampled, 10000)
df_test_debug = stratified_sample_ensure_labels(df_test_sampled, 2500)

# Extracting the features for the smaller subset
X_train_small = scaler.transform([extract_combined_features_debug(row['path'], verbose=False) for _, row in df_train_debug.i
y_train_small = df_train_debug['label'].values

X_test_small = scaler.transform([extract_combined_features_debug(row['path'], verbose=False) for _, row in df_test_debug.ite
y_test_small = df_test_debug['label'].values

# Training the Model SVM classifier on the smaller subset
svm_classifier_small = SVC(kernel='linear', probability=True, verbose=False)  # Enable probability estimates
svm_classifier_small.fit(X_train_small, y_train_small)

# Making predictions on a smaller subset
y_pred_small = svm_classifier_small.predict(X_test_small)

# Evaluating the classifier on the smaller subset
accuracy_small = accuracy_score(y_test_small, y_pred_small)
report_small = classification_report(y_test_small, y_pred_small)
conf_matrix_small = confusion_matrix(y_test_small, y_pred_small)
```

Figure 10: SVM Model Training and Hyperparameter Optimization

6.2 Logistic Regression Training

- **Overview:** Logistic Regression is a simple yet effective model for binary classification tasks.
- **Training Process:**
  - The model is trained on the same feature set as the SVM.
  - Regularization techniques are employed to prevent overfitting and improve model generalization.
- **Purpose:** Logistic Regression serves as a baseline model to compare the performance of more complex models.

```python
# Logistic Regression
print_current_time("Executing Logistic Regression")

logistic_classifier = LogisticRegression(max_iter=5000, verbose=1, solver='saga')
logistic_classifier.fit(X_train_scaled, y_train)

# Making predictions with Logistic Regression
y_pred_logistic = logistic_classifier.predict(X_test_scaled)

# Evaluating the Logistic Regression classifier
accuracy_logistic = accuracy_score(y_test, y_pred_logistic)
report_logistic = classification_report(y_test, y_pred_logistic)
conf_matrix_logistic = confusion_matrix(y_test, y_pred_logistic)
```

Figure 11: Logistic Regression Model Training

6.3 Model Evaluation Metrics Implementation

The author implemented key evaluation metrics to assess model performance, including accuracy, precision, recall, F1 Score, ROC curve, AUC, and confusion matrix. These metrics provided a comprehensive analysis of the classification models that highlights their strengths and areas for improvement.

```python
# Plot confusion matrix
def plot_confusion_matrix(conf_matrix, title):
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=["Fake", "Real"], yticklabels=["Fake", "Real"])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(title)
    plt.show()

# Plot ROC curve
def plot_roc_curve(y_true, y_score, title):
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(title)
    plt.legend(loc="lower right")
    plt.show()
```

Figure 12: Implementation of Model Evaluation Metrics: Accuracy, Precision, Recall, F1 Score, ROC Curve, and Confusion Matrix.

This section details the methods used to train and evaluate the models, highlighting the strengths and limitations of each approach. By using multiple evaluation metrics the author ensures assessment of model.

References