

Mood Recognition System

A PROJECT REPORT

Submitted By

Divyansh Bhatnagar
2000290140042

Ayushi Singhal
2000290140036

Tarunima Sharma
2000290140126

Akshay Singh
2000290140014

**Submitted in partial fulfilment of the
Requirements for the Degree of**

MASTER OF COMPUTER APPLICATIONS

**Under the Supervision of
Dr. Akash Rajak
Assistant Professor**



Submitted to

**DEPARTMENT OF COMPUTER APPLICATIONS
KIET Group of Institutions, Ghaziabad
Uttar Pradesh-201206
(13 JAN 2022)**

CERTIFICATE

Certified that **Divyansh Bhatnagar 2000290140042, Ayushi Singhal 2000290140036, Tarunima Sharma 2000290140126, Akshay Singh 2000290140014**, have carried out the project work having “**Mood Recognition System**” for Master of Computer Applications from Dr. A.P.J. Abdul Kalam Technical University (AKTU) (formerly UPTU), Technical University, Lucknow under my supervision. The project report embodies original work, and studies are carried out by the student himself / herself and the contents of the project report do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

Date:

Divyansh Bhatnagar 2000290140042

Ayushi Singhal 2000290140036

Tarunima Sharma 2000290140126

Akshay Singh 2000290140014

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date:

Dr. Akash Rajak
Assistant Professor
Department of Computer Applications
KIET Group of Institutions, Ghaziabad

Signature of Internal Examiner

Signature of External Examiner

Dr. Ajay Shrivastava
Head, Department of Computer Applications
KIET Group of Institutions, Ghaziabad

ABSTRACT

Our Mood Recognition System identifies emotions to the best of its capabilities depending on the internet and the hardware implemented in the system. communication that vary in complexity, intensity, and meaning. Purposed system depends upon human face as we know face also reflects the human brain activities or emotions.

The addition or absence of one or more facial actions may alter its interpretation. In addition, some facial expressions may have a similar gross morphology but indicate varied meaning for different expression intensities. In order to capture the subtlety of facial expression in non-verbal communication,

I will use an existing simulator which will be able to capture human emotions by reading or comparing mood expressions. This algorithm automatically extracts features and their motion information, discriminate subtly different facial expressions, and estimate expression intensity.

ACKNOWLEDGEMENTS

Success in life is never attained single handedly. My deepest gratitude goes to my thesis supervisor, **Dr. Akash Rajak** for his guidance, help and encouragement throughout my research work. Their enlightening ideas, comments, and suggestions.

Words are not enough to express my gratitude to Dr. Ajay Kumar Shrivastava, Professor and Head, Department of Computer Applications, for his insightful comments and administrative help at various occasions.

Fortunately, I have many understanding friends, who have helped me a lot on many critical conditions.

Finally, my sincere thanks go to my family members and all those who have directly and indirectly provided me moral support and other kind of help. Without their support, completion of this work would not have been possible in time. They keep my life filled with enjoyment and happiness.

Divyansh Bhatnagar

Ayushi Singhal

Tarunima Sharma

Akshay Singh

List of Chapters

Chapter 1 - Introduction

- 1.1 Overall description
- 1.2 Project Scope
- 1.3 Hardware / Software used in Project

Chapter 2 Feasibility Study

- 2.1 Technical feasibility
- 2.2 Operational Feasibility
- 2.3 Behavioral Feasibility
- 2.4 Operational Feasibility

Chapter 3 Database Design

- 3.1 Flow Chart
- 3.2 Use Case Diagram

Chapter 4 Form Design

- 4.1 Output Form (Screenshot)

Chapter 5 Coding

- Whole Source code

Chapter 6 Testing

- 6.1 Unit Testing
- 6.2 Integration Testing
- 6.3 Software Verification and Validation
- 6.4 Black Box Testing
- 6.5 White Box Testing
- 6.6 System Testing
- 6.7 Test Cases

Bibliography

Chapter 1

Introduction

1.1 Overall Description

1.1.1 Product Perspective

An emotion recognition system can detect the emotion condition of a person either from his image or speech information. In this scope, an audio-visual emotion recognition system requires to evaluate the emotion of a person from his speech and image information together.

1.1.2 Product Features

The software described in this SRS will be used to detect people's emotions. This project can be used in several areas that like to measure customer satisfaction in a marketing platform, helping advertisers to sell products more effectively.

1.2 Project Scope

Modern day security systems rely heavily on bioinformatics, like speech, fingerprint, facial images and so on. Besides, determination of a user's emotional state with facial and voice analysis plays a fundamental part in human-machine interaction (HMI) systems, since it employs non-verbal cues to estimate the user's emotional state. This software system will be able to perform emotion recognition from audio, video and audio-visual video. With the easy-to-use user-interface of the system, the user can either record instant video/real time or upload an existing video to the system and perform emotion recognition. This system allows big corporate companies to measure customer satisfaction and perform the necessary analysis.

1.3 HARDWARE/SOFTWARE USED IN PROJECT

1.3.1 Hardware Specification

- Central Processing Unit (CPU) - Intel Core i5 6th gen or AMD processor equivalent
- RAM - 8 GB minimum, 16 GB or higher is recommended.
- Graphics Processing Unit (GPU) - NVIDIA GeForce GTX 960 or higher
- Inbuilt Camera or Webcam Support
- Operating System (OS) - Ubuntu or Microsoft Windows 10
- Storage - 20 GB

1.3.2 Software Specification

- Python 3 or above
- Library - open cv

Chapter 2

2 FEASIBILITY STUDY

A feasibility study is a high-level capsule version of the entire System analysis and design Process. The study begins by classifying the problem definition. Feasibility is to determine if it's worth doing. Once an acceptance problem definition has been generated, the analyst develops a logical model of the system. A search for alternatives is analyzed carefully. There are 3 parts in feasibility study.

2.1 TECHNICAL STUDY

his involves questions such as whether the technology needed for the system exists, how difficult it will be to build, and whether the firm has enough experience using that technology. The assessment is based on outline design of system requirements in terms of input, processes, output, fields, programs, and procedures. This can be qualified in terms of volume of data, trends, frequency of updating in-order to introduce the technical system. The application is the fact that it has been developed on windows 10 platform and a high configuration of 8 GB RAM on Intel Pentium Dual core processor. This is technically feasible. The technical feasibility assessment is focused on gaining an understanding of the present technical resources of the organization and their applicability to the expected needs of the proposed system. It is an evaluation of the hardware and software and how it meets the need of the proposed system.

2.2 OPERATIONAL STUDY

Operational feasibility is the measure of how well a proposed system solves the problems and takes advantage of the opportunities identified during scope definition and how it satisfies the requirements identified in the requirements analysis phase of system development. The operational feasibility assessment focuses on the degree to which the proposed development projects fits in with the existing business environment and objectives with regard to development schedule, delivery date, corporate culture and existing business processes. To ensure success, desired operational outcomes must be imparted during design and development. These include such design-dependent parameters as reliability, maintainability, supportability, usability, producibility, disposability, sustainability, affordability, and others. These parameters are required to be considered at the early stages of design if desired operational behaviors are to be realized. A system design and development require appropriate and timely application of engineering and management efforts to meet the previously mentioned parameters. A system may serve its intended purpose most effectively when its technical and operating characteristics are engineered into the design. Therefore, operational feasibility is a critical aspect of systems engineering that needs to be an integral part of the early design phases.

2.3 BEHAVIORAL STUDY

Establishing the cost-effectiveness of the proposed system i.e., if the benefits do not outweigh the costs, then it is not worth going ahead. In the fast-paced world today there is a great need of online social networking facilities. Thus, the benefits of this project in the current scenario make it economically feasible. The purpose of the economic feasibility assessment is to determine the positive economic benefits to the organization that the proposed system will provide. It includes quantification and identification of all the benefits expected. This assessment typically involves a cost/benefits analysis.

Chapter 3

3.1 FLOW CHART

Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing. The process of drawing a flowchart for an algorithm is known as “flowcharting”.

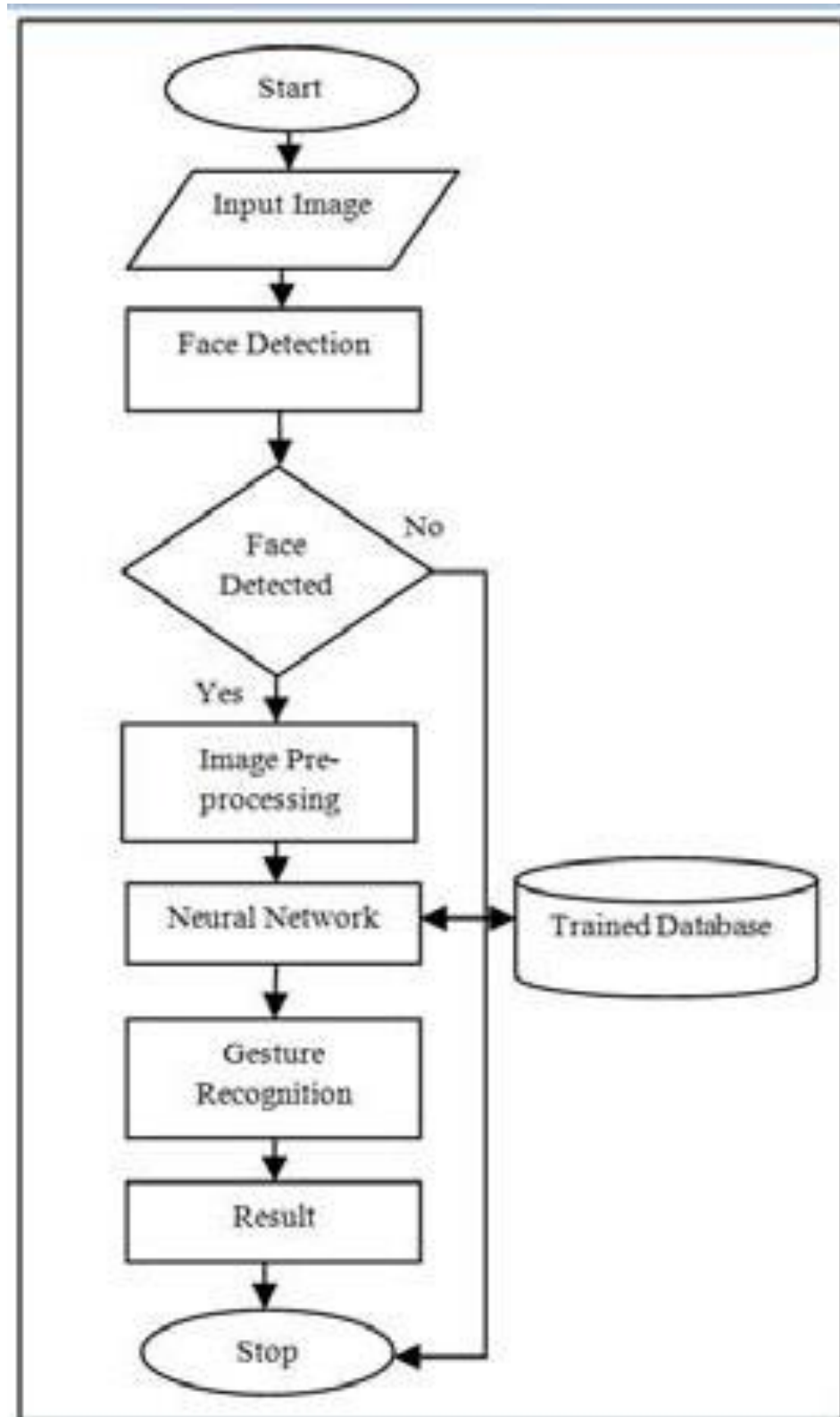
Basic Symbols used in Flowchart Designs

- 1. Terminal:** The oval symbol indicates Start, Stop and Halt in a program’s logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.
- 2. Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.
- 3. Processing:** A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.
- 4. Decision:** Diamond symbol represents a decision point. Decision based operations such as yes/no

question or true/false are indicated by diamond in flowchart.

5. Connectors: Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.

6. Flow lines: Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.



3.2 USE CASE DIAGRAM

In the Unified Modeling Language (UML), a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system. To build one, you'll use a set of specialized symbols and connectors. An effective use case diagram can help your team discuss and represent:

- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve

Use case Diagram Components

To answer the question, "What is a use case diagram?" you need to first understand its building blocks. Common components include:

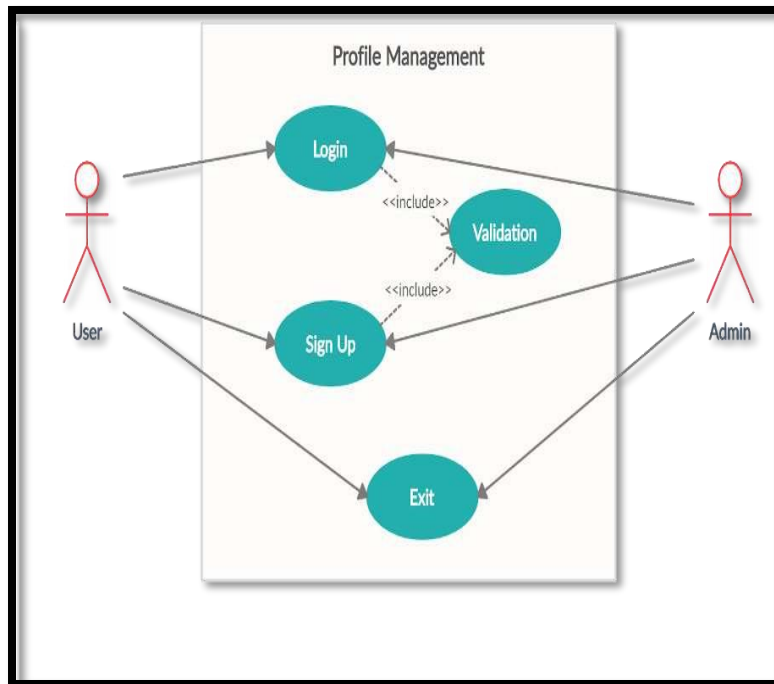
- **Actors:** The users that interact with a system. An actor can be a person, an organization, or an outside system that interacts with your application or system. They must be external objects that produce or consume data.
- **System:** A specific sequence of actions and interactions between actors and the system. A system may also be referred to as a scenario.
- **Goals:** The result of most use cases. A successful diagram should describe the activities and variants used to reach the goal.

Use case diagram symbols and notation

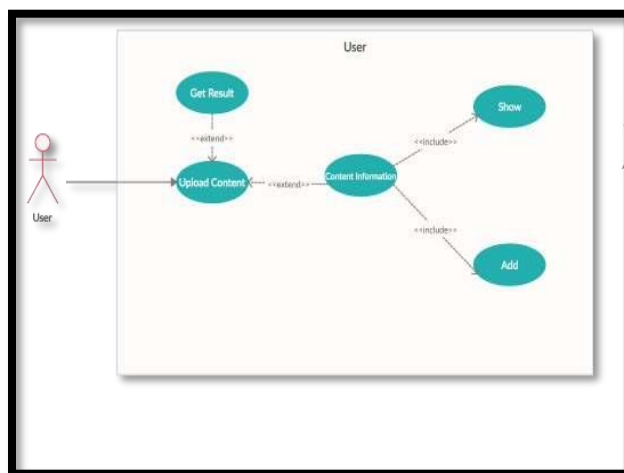
The notation for a use case diagram is straightforward and doesn't involve as many types of symbols as other UML diagrams.

Use cases: Horizontally shaped ovals that represent the different uses that a user might have.

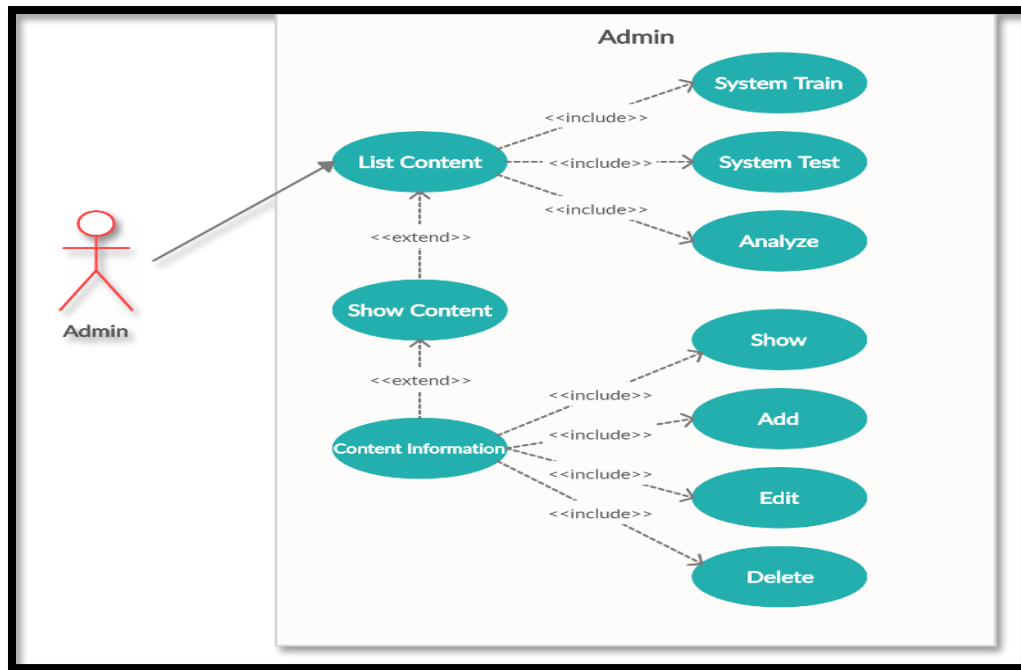
- **Actors:** Stick figures that represent the people employing the use cases.
- **Associations:** A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- **System boundary boxes:** A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system. For example, Psycho Killer is outside the scope of occupations in the chainsaw example found below.
- **Packages:** A UML shape that allows you to put different elements into groups. Just as with component diagrams, these groupings are represented as file folders.



3.1.1 Profile Management Use Case



3.1.2 User Use Case

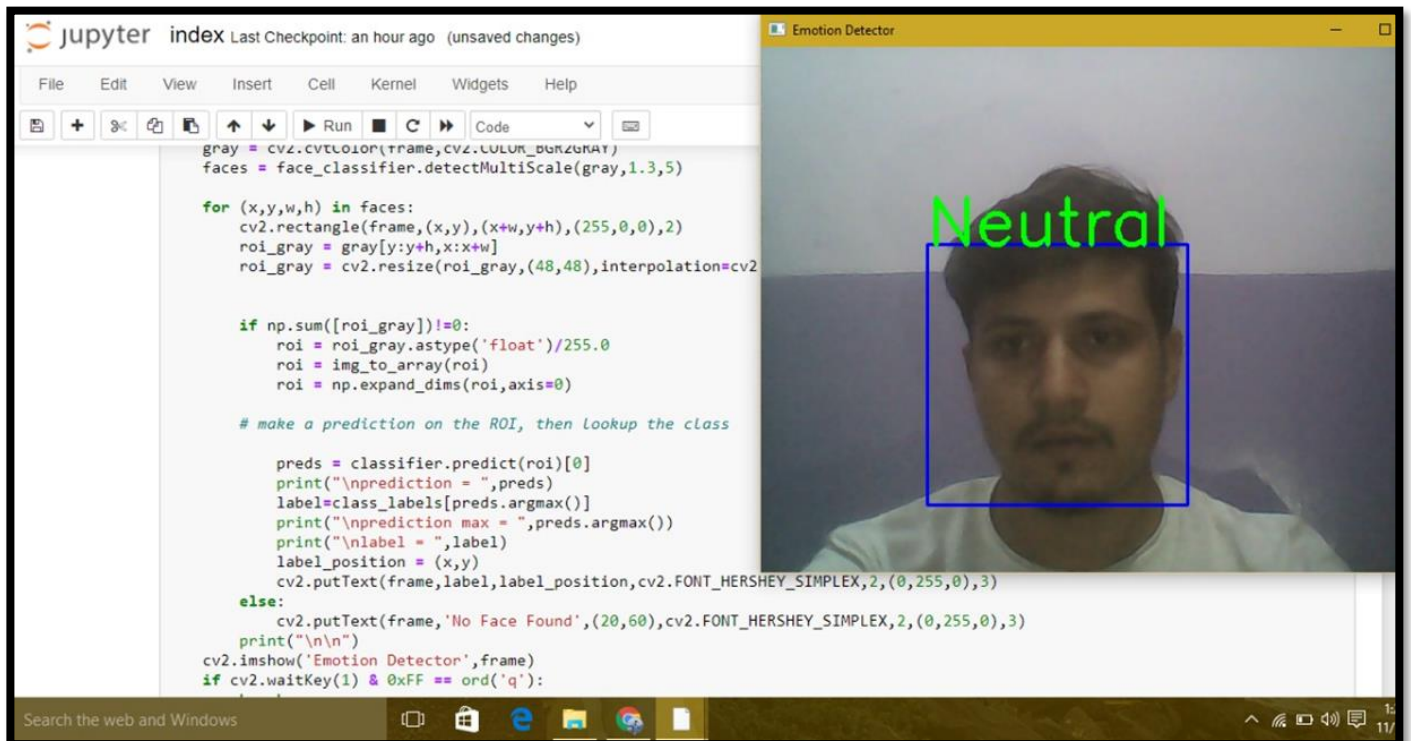


3.1.3 Admin Use Case

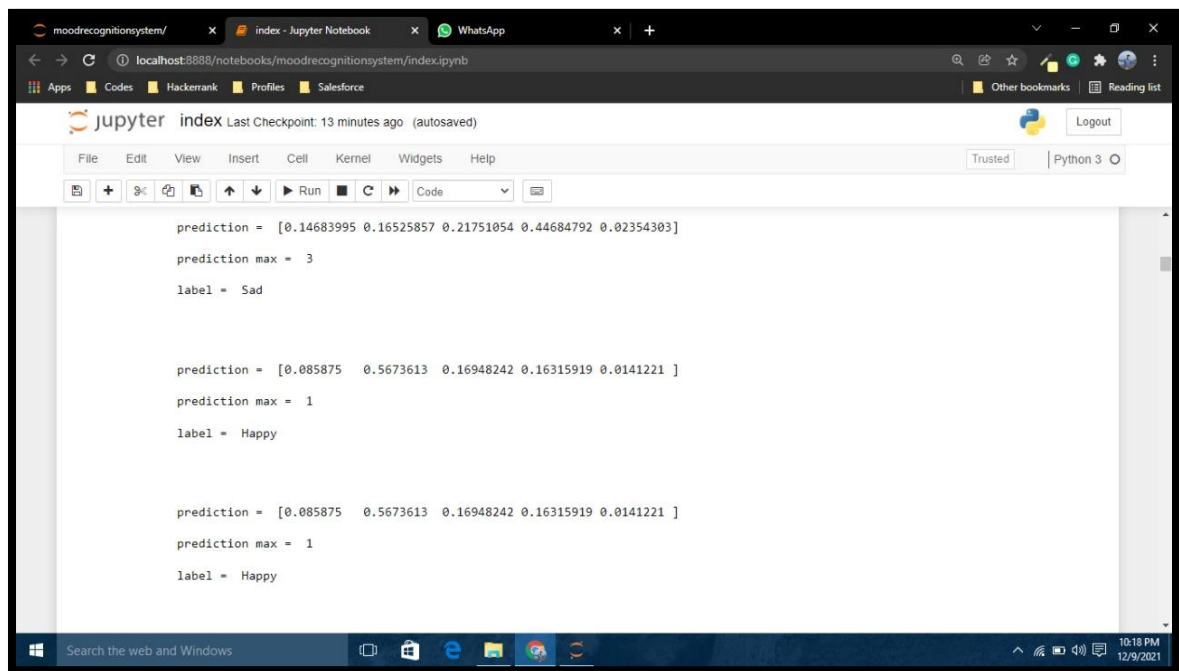
Chapter 4

Design

4.1 Output 1 – System Look



4.2 Output 2 – Command Line Output



The screenshot shows a Jupyter Notebook interface in a web browser. The browser tabs include 'moodrecognitionssystem/', 'index - Jupyter Notebook', and 'WhatsApp'. The address bar shows 'localhost:8888/notebooks/moodrecognitionssystem/index.ipynb'. The Jupyter interface has a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and code execution. The notebook title is 'index' with a 'Last Checkpoint: 13 minutes ago (autosaved)' status. The code cell contains three sets of predictions for mood recognition, each with a list of probabilities, the maximum probability index, and the corresponding label.

```
prediction = [0.14683995 0.16525857 0.21751054 0.44684792 0.02354303]
prediction max = 3
label = Sad

prediction = [0.085875 0.5673613 0.16948242 0.16315919 0.0141221 ]
prediction max = 1
label = Happy

prediction = [0.085875 0.5673613 0.16948242 0.16315919 0.0141221 ]
prediction max = 1
label = Happy
```

4.3 Output-3 Emotion Command Line Output

```
In [5]: print(class_labels[preds.argmax()])
```

```
Sad
```

Chapter 5

Coding

5.1: Main Source Code

```
from keras.models import load_model
from time import sleep
from keras.preprocessing.image import img_to_array
from keras.preprocessing import image
import cv2
import numpy as np

face_classifier =
cv2.CascadeClassifier('./haarcascade_frontalface_default.xml')
classifier =load_model('./Emotion_Detection.h5')

class_labels = ['Angry','Happy','Neutral','Sad','Surprise']

cap = cv2.VideoCapture(0)

while True:
    # Grab a single frame of video
    ret, frame = cap.read()
    labels = []
    gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
    faces = face_classifier.detectMultiScale(gray,1.3,5)

    for (x,y,w,h) in faces:
```

```

cv2.rectangle(frame,(x,y),(x+w,y+h),(255,0,0),2)
roi_gray = gray[y:y+h,x:x+w]
roi_gray =
cv2.resize(roi_gray,(48,48),interpolation=cv2.INTER_AREA)
if np.sum([roi_gray])!=0:
    roi = roi_gray.astype('float')/255.0
    roi = img_to_array(roi)
    roi = np.expand_dims(roi,axis=0)

    # make a prediction on the ROI, then lookup the class
    preds = classifier.predict(roi)[0]
    print("\nprediction = ",preds)
    label=class_labels[preds.argmax()]
    print("\nprediction max = ",preds.argmax())
    print("\nlabel = ",label)
    label_position = (x,y)

cv2.putText(frame,label,label_position,cv2.FONT_HERSHEY_SIMPL
EX,2,(0,255,0),3)
else:
    cv2.putText(frame,'No Face
Found',(20,60),cv2.FONT_HERSHEY_SIMPLEX,2,(0,255,0),3)
    print("\n\n")
cv2.imshow('Emotion Detector',frame)
if cv2.waitKey(1) & 0xFF == ord('q'):
    break

cap.release()
cv2.destroyAllWindows()

```

Chapter 6

Testing

6.1 Unit Testing

In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method. Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process. It forms the basis for component testing. Ideally, each test case is independent from the others. Substitutes such as method stubs, mock objects, fakes, and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended.

6.1.2 Benefits

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A

unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits.

1. **Find problems early:** Unit testing finds problems early in the development cycle. In test-driven development (TDD), which is frequently used in both extreme programming and scrum, unit tests are created before the code itself is written. When the tests pass, that code is considered complete. The same unit tests are run against that function frequently as the larger code base is developed either as the code is changed or via an automated process with the build. If the unit tests fail, it is a bug either in the changed code or the tests themselves. The unit tests then allow the location of the fault or failure to be easily traced. Since the unit tests alert the development team of the problem before handing the code off to testers or clients, it is still early in the development process.

2. **Facilitates Change:** Unit testing allows the programmer to refactor code or upgrade system libraries later, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified. Unit tests detect changes which may break a design contract.

3. **Simplifies Integration:** Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts

of a program first and then testing the sum of its parts, integration testing becomes much easier.

4. **Documentation:** Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit, and how to use it, can look at the unit tests to gain a basic understanding of the unit's interface (API). Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviors that are to be trapped by the unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in development.

6.2: INTEGRATION TESTING

Integration testing (sometimes called integration and testing, abbreviated I&T) is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

6.2.1 Purpose

The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items. These "design items", i.e., assemblages (or groups of units), are exercised through their interfaces using black-box testing, success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface. Test cases are constructed to test whether all the components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e., unit testing. The overall idea is a "building block" approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages. Software integration testing is performed according to the software development life cycle (SDLC) after module and functional tests. The cross-dependencies for software integration testing are: schedule for integration testing, strategy and selection of the tools used for integration, define the cyclomatic complexity of the software and software architecture, reusability of modules and life-cycle and versioning management. Some different types of integration testing are big-bang, top-down, and bottom-up, mixed (sandwich) and risky-hardest. Other Integration Patterns [2] are collaboration integration,

backbone integration, layer integration, client-server integration, distributed services integration and high-frequency integration.

6.2.1.1 Big Bang

In the big-bang approach, most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing. This method is very effective for saving time in the integration testing process. However, if the test cases and their results are not recorded properly, the entire integration process will be more complicated and may prevent the testing team from achieving the goal of integration testing. A type of big-bang integration testing is called "usage model testing" which can be used in both software and hardware integration testing. The basis behind this type of integration testing is to run user-like workloads in integrated user-like environments. In doing the testing in this manner, the environment is proofed, while the individual components are proofed indirectly through their use. Usage Model testing takes an optimistic approach to testing because it expects to have few problems with the individual components. The strategy relies heavily on the component developers to do the isolated unit testing for their product. The goal of the strategy is to avoid redoing the testing done by the developers, and instead flesh-out problems caused by the interaction of the components in the environment. For integration testing, Usage Model testing can be more efficient and provides better test coverage than traditional focused functional integration testing. To be

more efficient and accurate, care must be used in defining the user-like workloads for creating realistic scenarios in exercising the environment. This gives confidence that the integrated environment will work as expected for the target customers.

6.2.1.2 Top-down And Bottom-up

Bottom-up testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher-level components. The process is repeated until the component at the top of the hierarchy is tested. All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower-level integrated modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage. Top-down testing is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module. Sandwich testing is an approach to combine top down testing with bottom up testing.

6.3: SOFTWARE VERIFICATION AND VALIDATION

6.3.1 Introduction

In software project management, software testing, and software engineering, verification and validation (V&V) is the process of checking that a software system meets specifications and that it fulfills its intended purpose. It may also be referred to as software quality control. It is normally the responsibility of software testers as part of the software development lifecycle. Validation checks that the product design satisfies or fits the intended use (high-level checking), i.e., the software meets the user requirements. This is done through dynamic testing and other forms of review. Verification and validation are not the same thing, although they are often confused. Boehm succinctly expressed the difference between

- Validation: Are we building the right product?
- Verification: Are we building the product right?

According to the Capability Maturity Model (CMMI-SW v1.1)

Software Verification: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Software Validation: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

In other words, software verification is ensuring that the product has been built according to the requirements and design specifications, while software validation ensures that the product meets the user's needs, and that the specifications were correct in the first place. Software verification ensures that "you built it right". Software validation ensures that "you built the right thing". Software validation confirms that the product, as provided, will fulfill its intended use.

From Testing Perspective

- Fault – wrong or missing function in the code.
- Failure – the manifestation of a fault during execution.
- Malfunction – according to its specification the system does not meet its specified functionality

Both verification and validation are related to the concepts of quality and of software quality assurance. By themselves, verification and validation do not guarantee software quality; planning, traceability, configuration management and other aspects of software engineering are required. Within the modeling and simulation (M&S) community, the definitions of verification, validation and accreditation are similar:

- M&S Verification is the process of determining that a computer model, simulation, or federation of models

and simulations implementations and their associated data accurately represent the developer's conceptual description and specifications.

- M&S Validation is the process of determining the degree to which a model, simulation, or federation of models and simulations, and their associated data are accurate representations of the real world from the perspective of the intended use(s).

6.3.2 Classification of Methods

In mission-critical software systems, where flawless performance is absolutely necessary, formal methods may be used to ensure the correct operation of a system.

However, often for non- mission-critical software systems, formal methods prove to be very costly and an alternative method of software V&V must be sought out. In such cases, syntactic methods are often used.

6.3.3 Test Cases

A test case is a tool used in the process. Test cases may be prepared for software verification and software validation to determine if the product was built according to the requirements of the user. Other methods, such as reviews, may be used early in the life cycle to provide for software validation.

6.4: Black-Box Testing

Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings. This method of test can be applied virtually to every level of software testing: unit, integration, system and acceptance. It typically comprises most if not all higher-level testing but can also dominate unit testing as well.

6.4.1 Test Procedures

Specific knowledge of the application's code/internal structure and programming knowledge in general is not required. The tester is aware of what the software is supposed to do but is not aware of how it does it. For instance, the tester is aware that a particular input returns a certain, invariable output but is not aware of how the software produces the output in the first place.

6.4.2 Test Cases

Test cases are built around specifications and requirements, i.e., what the application is supposed to do. Test cases are generally derived from external descriptions of the software, including specifications, requirements and design parameters. Although the tests used are primarily functional in nature, non-functional tests may also be used. The test designer selects both valid and invalid inputs and determines the correct output, often with the help of an oracle or a previous result that is known to be good, without any knowledge of the test object's internal structure.

6.5 : White-Box Testing

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT). White-box testing can be applied at the unit, integration and system levels of the software testing process. Although traditional testers tended to think of white-box testing as being done at the unit level, it is used for integration and system testing more frequently today. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it has the potential to miss unimplemented parts of the specification or missing requirements.

6.5.1 Levels

1) Unit testing : White-box testing is done during unit testing to ensure that the code is working as intended, before any integration happens with previously tested code. White box testing during unit testing catches any defects

early on and aids in any defects that happen later on after the code is integrated with the rest of the application and therefore prevents any type of errors later on.

2) Integration testing : White-box testing at this level are written to test the interactions of each interface with each other. The Unit level testing made sure that each code was tested and working accordingly in an isolated environment and integration examines the correctness of the behavior in an open environment through the use of white-box testing for any interactions of interfaces that are known to the programmer.

3) Regression testing : White-box testing during regression testing is the use of recycled white-box test cases at the unit and integration testing levels.

6.5.2 Procedures

White-box testing's basic procedures involves the tester having a deep level of understanding of the source code being tested. The programmer must have a deep understanding of the application to know what kinds of test cases to create so that every visible path is exercised for testing. Once the source code is understood then the source code can be analyzed for test cases to be created.

These are the three basic steps that white-box testing takes in order to create test cases:

- Input involves different types of requirements, functional specifications, detailed designing of documents, proper source code, security

specifications. This is the preparation stage of white box testing to layout all of the basic information.

- Processing involves performing risk analysis to guide whole testing process, proper test plan, execute test cases and communicate results. This is the phase of building test cases to make sure they thoroughly test the application the given results are recorded accordingly.
- Output involves preparing final report that encompasses all of the above preparations and results.

6.5.3 Advantages

White-box testing is one of the two biggest testing methodologies used today. It has several major advantages:

- Side effects of having the knowledge of the source code is beneficial to thorough testing.
- Optimization of code by revealing hidden errors and being able to remove these possible defects.
- Gives the programmer introspection because developers carefully describe any new implementation.
- Provides traceability of tests from the source, allowing future changes to the software to be easily captured in changes to the tests.
- White box testing give clear, engineering-based, rules for when to stop testing.

6.5.5 Disadvantages

Although white-box testing has great advantages, it is not perfect and contains some disadvantages:

- White-box testing brings complexity to testing because the tester must have knowledge of the program, including being a programmer. White-box testing requires a programmer with a high level of knowledge due to the complexity of the level of testing that needs to be done.
- On some occasions, it is not realistic to be able to test every single existing condition of the application and some conditions will be untested.
- The tests focus on the software as it exists, and missing functionality may not be discovered.

6.6: SYSTEM TESTING

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black-box testing, and as such, should require no knowledge of the inner design of the code or logic. As a rule, system testing takes, as its input, all of the "integrated" software components that have passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limited type of testing; it seeks to detect defects both within

the "inter-assemblages" and also within the system as a whole.

System testing is performed on the entire system in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS). System testing tests not only the design, but also the behavior and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specification(s).

6.7: Test Cases

TEST CASE TEMPLATE							
S.NO.	INPUT	ANGRY	HAPPY	NEUTRAL	SAD	SURPRISE	OUTPUT
1	SITTING IN FRONT OF CAMERA	0.14683995	0.16525857	0.21751054	0.44684792	0.02354303	SAD
2	SITTING IN FRONT OF CAMERA	0.085875	0.5673613	0.16948242	0.161315919	0.0141221	HAPPY
3	SITTING IN FRONT OF CAMERA	0.64683995	0.16525857	0.11751054	0.14684792	0.02354303	ANGRY
4	SITTING IN FRONT OF CAMERA	0.085875	0.5673613	0.16948242	0.161315919	0.0141221	HAPPY
5	SITTING IN FRONT OF CAMERA	0.14683995	0.16525857	0.21751054	0.44684792	0.02354303	SAD
6	SITTING IN FRONT OF CAMERA	0.14683995	0.16525857	0.41751054	0.14684792	0.02354303	NEUTRAL

Bibliography

- <https://pypi.org/project/keras/>
- https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml
- <https://numpy.org/>
- <https://www.wisegeek.com/what-is-emotional-intelligence.htm>