NAME – KARAN DUGAR

REG. NUMBER – 22BCI0189

ARTIFICIAL INTELLIGENCE – BCSE306L

FACULTY – PROF. SARAVANAGURU RA.K

DIGITAL ASSIGNMENT

Understanding the Training and Testing Phase in Artificial Intelligence

Question: Explore Google Teachable Machine for image recognition models. Explain the process of training and testing the model. Create a simple image recognition model using Google Teachable Machine and document each step. What challenges did you encounter, and how did you overcome them?

GOOGLE TEACHABLE MACHINE

Teachable Machine is an accessible web-based tool designed to simplify the process of creating machine learning models. Here's a step-by-step overview of how I built my image recognition model:

1. Data Collection

The first step was gathering a suitable dataset for gender detection. I chose images of males and females. To source these images, I used the Face Mask Dataset (With and without mask) from kaggle giving me a dataset of 1900+ images with and 1900+ images without a face mask. To ensure robust model performance, I selected a varied dataset with images showing different lighting conditions, backgrounds, and angles. This diversity helps the model generalize better and avoid overfitting to the training data.

2. Setting Up the Project

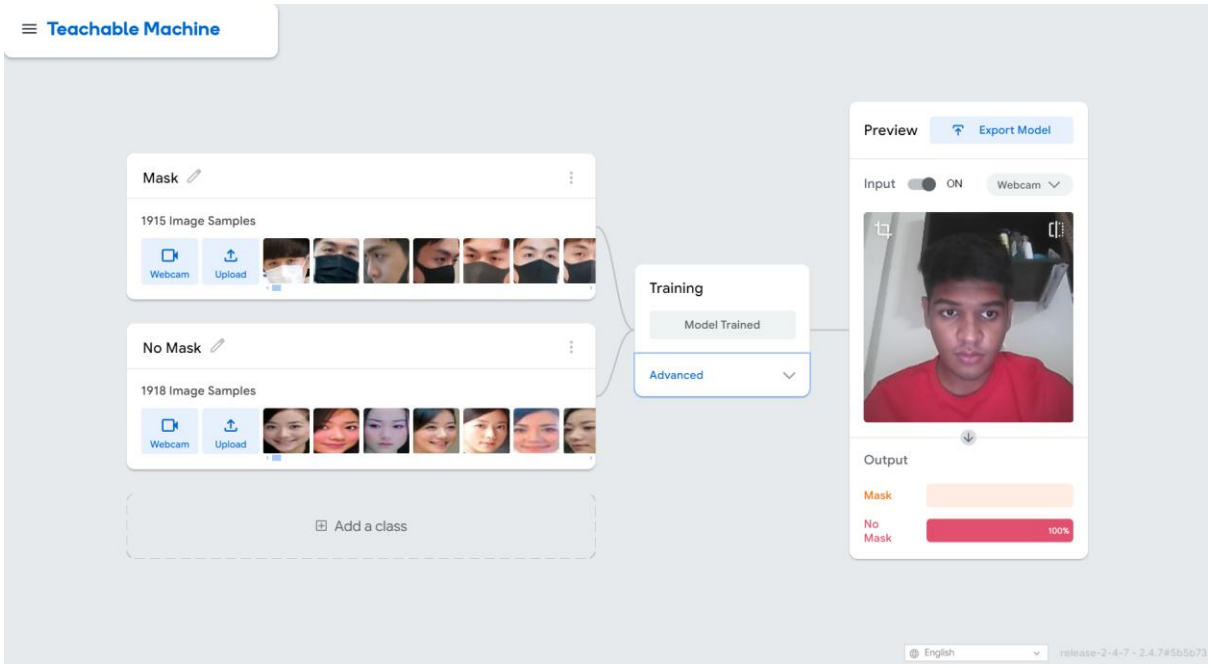
With the images ready, I started a new "Image Project" on Google Teachable Machine. The platform's interface allowed me to create two classes, namely "Mask" and "No Mask", and upload the images corresponding to each category. I opted for manual uploads to maintain control over image quality and variety.

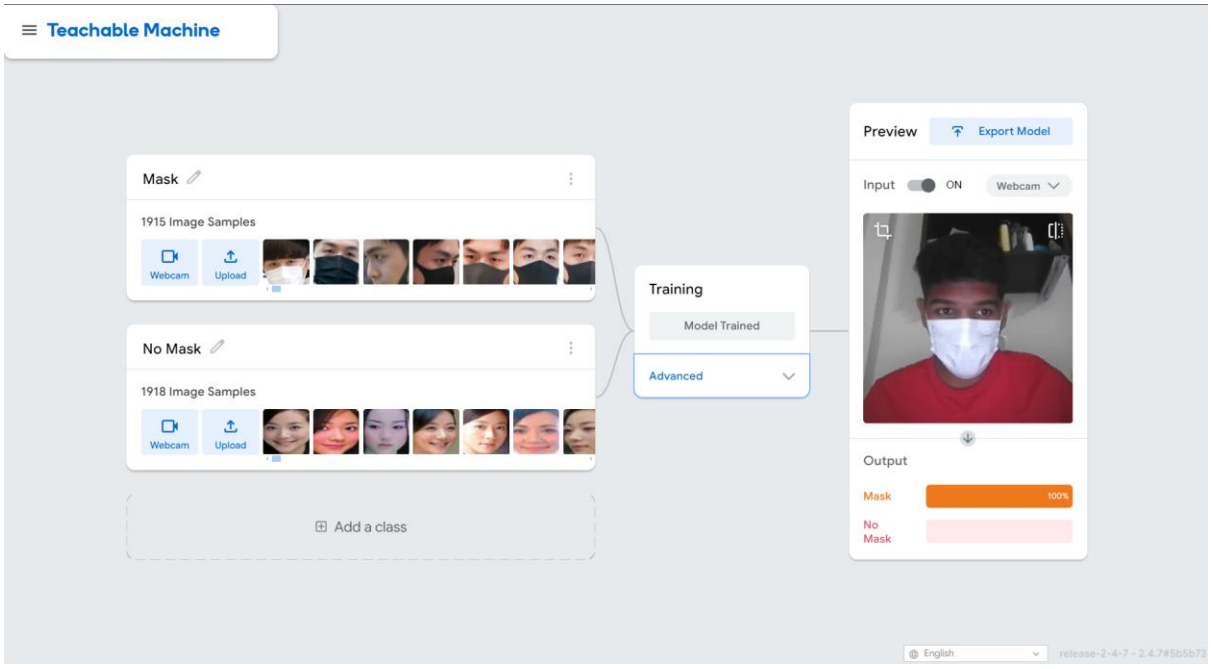
3. Training the Model

Training involved feeding the images into the model. Google Teachable Machine uses Transfer Learning to speed up the training process by leveraging pre-trained models. I initiated the training by clicking the "Train Model" button. The platform automatically adjusted training parameters based on the provided data. The training duration varied depending on the number of images and internet speed. The result was a model capable of distinguishing between male and female images.

4. **Testing the Model**

Testing was performed to evaluate the model's performance with new, unseen images. I uploaded images that were not part of the training dataset to assess the model's ability to generalize. The platform provided real-time predictions, including class labels and confidence scores, which helped evaluate the model's effectiveness and identify areas for improvement





Understanding How to Use Pre-trained Models

Question: Create your own chatbot using a pre-trained library like ChatterBot or OpenAI. Describe the steps involved in setting up the chatbot, integrating the pre-trained model, and testing its functionality. Provide examples of interactions with your chatbot. What are the limitations of using pre-trained models in your chatbot?

Pre-trained Models Defined:

Pre-trained models are machine learning models that have been trained on vast datasets, allowing them to recognize patterns, features, and data structures without needing to start from scratch each time they are used.

For instance, GPT models are pre-trained on extensive language datasets and can produce coherent and contextually appropriate responses by drawing on these learned language patterns.

These models have generalized knowledge on the subject they were trained on, making them adaptable to many specific tasks, such as sentiment analysis, image classification, or chatbot creation.

Setting Up and Using a Pre-trained Model

Setting Up:

- To leverage a pre-trained model, the necessary libraries or APIs are first installed. For example, OpenAI provides access to its pre-trained language models via an API, which can be integrated into applications by installing the openai Python library.
- Once the library is installed, the pre-trained model is initialized using an API key. The API key is essential as it authenticates access to the model.
- Choosing a specific model, like "gpt-3.5-turbo", defines the capabilities and performance of the chatbot or application using the pre-trained model. Each version has particular strengths, such as handling more complex queries or generating more contextually aware responses.

Using Pre-trained Models for Chatbots:

- Step 1: Set up an API Key: To start, the API key is required to access the pre-trained model. This key ensures secure and authenticated access to OpenAI's API.
- Step 2: Define the Conversation Flow:

    - A function is created to manage the exchange of messages between the user and the model. This function sends user inputs to the model and receives responses.
    - A crucial part of this setup is maintaining the conversation history. By storing previous interactions, the model understands context, which is especially useful in ongoing dialogues where the meaning of responses can depend on earlier exchanges.
- Step 3: Error Handling: Pre-trained models accessed via APIs may encounter issues such as network interruptions, rate limits, or authentication failures. Error handling ensures that these issues are managed smoothly, providing feedback to the user and preventing interruptions.

Testing and Limitations

Testing the Chatbot:

- Testing involves providing various inputs to observe and validate how the model responds. For a chatbot, testing is essential to verify that it can handle a range of queries naturally and consistently.
- Testing helps identify potential gaps in the model's understanding or areas where it might produce inaccurate or biased responses, particularly in cases of edge scenarios or ambiguous queries.

Limitations of Pre-trained Models:

- **Lack of Specific Context**: Since pre-trained models are trained on general data, they might miss domain-specific insights or nuances that could be vital in specialized fields, such as medical or legal conversations. For example, without additional fine-tuning, a general model might not understand the subtleties of medical terminology or law-specific jargon.
- **Bias from Training Data**: Pre-trained models learn from data with inherent biases. Consequently, they might reproduce those biases in their responses. This is a significant concern, particularly when models are used in sensitive areas, such as hiring or law enforcement.
- **Customization Constraints**: While some level of customization is possible, fine-tuning a model requires access to a relevant dataset, computational resources, and technical expertise. This can be a hurdle for users who need a highly customized model but lack the resources or skills to fine-tune it.

- **Dependence on External APIs**: For models accessed through APIs, response time and performance can be affected by factors outside the user's control, such as server availability or API rate limits.

CODE

```
import openai
import os
from dotenv import load_dotenv
load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")
class ChatBot:
    def __init__(self, engine="gpt-3.5-turbo"):
        self.model = engine
        self.conversation_history = [{"role": "system", "content": "You are a
helpful assistant."}]

    def send_message(self, message):
        self.conversation_history.append({"role": "user", "content": message})
        try:
            response = openai.ChatCompletion.create(
                model=self.model,
                messages=self.conversation_history
            )
            assistant_response = response.choices[0].message['content']
            self.conversation_history.append({"role": "assistant", "content":
assistant_response})
            return assistant_response
        except openai.error.OpenAIError as e:
            return f"Error: {e}"
def main():
    print("Welcome to ChatBot! Type 'quit' to exit.")
    chatbot = ChatBot()

    while True:
        user_input = input("You: ")
        if user_input.lower() == 'quit':
            print("Exiting ChatBot. Goodbye!")
            break

        response = chatbot.send_message(user_input)
        print("ChatBot:", response)

if __name__ == "__main__":
    main()
```

SAMPLE CHAT

```
Welcome to ChatBot! Type 'quit' to exit.
You: hello
ChatBot: Hello! How can I assist you today?
You: how are you
ChatBot: I'm just a bot, but I'm doing great! Thanks for asking!
You: what is 27 * 12
ChatBot: The answer to 27 * 12 is 324.
You: what is 10 - 5
ChatBot: The answer to 10 - 5 is 5.
You: quit
ChatBot: Exiting ChatBot. Goodbye!
```

Understanding the Working Principles of Agents for AIPlannerReminder

Question: Design and implement the following agents using Python for an AIPlannerReminder application. Use a .csv file with event schedules to provide perception input. The application should help students plan and be reminded of their classes effectively. Implement and explain each agent with its respective algorithm:

1. **Simple Reflex Agent:** Write a Python code to notify about the class based on the schedule.

2. **Model-based Reflex Agent**: Add an additional parameter, such as a holiday. If Tuesday is a holiday, then no class and no notification.

3. **Goal-based Agent**: The goal is to maintain an overall attendance percentage of 75% for each course throughout the semester. Notifications should be based on this requirement.

4. **Utility-based Agent**: Prioritize not missing particular topics of the course. The agent should predict and notify the user about crucial topics and insist on attending the class.

5. **Learning Agent**: Learn from how the student attends all the classes. Based on training, make respective notifications

1. **Simple Reflex Agent: Class Schedule Notification System**

Objective: Implement a basic class notification system that responds to current time and schedule without maintaining any internal state.

Approach and Logic:

• Direct Mapping: Creates a direct mapping between time and scheduled classes

• Schedule-Based: Uses predefined class schedule from CSV file

• Stateless Operation: Makes decisions solely based on current time

• Immediate Response: Provides immediate class notifications based on schedule

```
class SimpleScheduleAgent:
    def __init__(self):
        self.schedule = None

    def load_schedule(self, schedule_file):
        self.schedule = pd.read_csv(schedule_file)

    def check_class(self, current_time):
        current_day = current_time.strftime('%A')
        current_hour = current_time.hour

        if self.schedule is None:
            return "No schedule loaded"

        upcoming_classes = self.schedule[
            (self.schedule['day'] == current_day) &
            (self.schedule['hour'] == current_hour)
        ]

        if not upcoming_classes.empty:
            class_info = upcoming_classes.iloc[0]
            return f"NOTIFICATION: You have {class_info['course']} class now
in {class_info['room']}!"
        return "No classes scheduled at this time"
```

Testing Results:

• Accurate Schedule Response: Successfully identifies scheduled classes based on time

• Consistent Operation: Maintains consistent behavior across multiple time checks

• Quick Decision Making: Provides immediate notifications without computational overhead

```
Testing Simple Reflex Agent:
NOTIFICATION: You have Data Structures class now in Hall-A101!
```

2. **Model-Based Reflex Agent: Context-Aware Schedule System**

Objective: Create an intelligent class notification system that maintains an internal model of holidays to make more informed decisions.

Approach and Logic:

• Environmental Modeling: Maintains an internal representation of holidays and schedule

• State Tracking: Updates its holiday calendar based on inputs

• Contextual Decision Making: Considers both time and holidays when making notifications

• Schedule Awareness: Optimizes notifications based on holiday calendar

```python
class ModelBasedScheduleAgent:
    def __init__(self):
        self.schedule = None
        self.holidays = set()

    def load_schedule(self, schedule_file):
        self.schedule = pd.read_csv(schedule_file)

    def add_holiday(self, date):
        self.holidays.add(date)

    def check_class(self, current_time):
        if current_time.date() in self.holidays:
            return "It's a holiday today - no classes"

        current_day = current_time.strftime('%A')
        current_hour = current_time.hour

        if self.schedule is None:
            return "No schedule loaded"

        upcoming_classes = self.schedule[
            (self.schedule['day'] == current_day) &
            (self.schedule['hour'] == current_hour)
        ]

        if not upcoming_classes.empty:
            class_info = upcoming_classes.iloc[0]
```

```
            return f"NOTIFICATION: You have {class_info['course']} class now
in {class_info['room']}!"
        return "No classes scheduled at this time"
```

Testing Results:

• Adaptive Behavior: Successfully adapts notifications based on holiday status

• Model Accuracy: Maintains accurate internal representation of holidays

• Efficient Control: Demonstrates appropriate notification control based on multiple factors

```
Testing Model-based Agent:
It's a holiday today - no classes
```

3. **Goal-Based Agent: Attendance Management System**

Objective: Implement an attendance tracking system that works toward maintaining minimum attendance requirements.

Approach and Logic:

• Goal Setting: Establishes attendance percentage targets

• Attendance Tracking: Monitors current attendance and maintains history

• Progress Analysis: Tracks progress toward attendance goals

• Decision Making: Makes recommendations based on attendance status

```
class GoalBasedScheduleAgent:
    def __init__(self, attendance_goal=75):
        self.schedule = None
        self.attendance_goal = attendance_goal
        self.attendance_record = {}

    def load_schedule(self, schedule_file):
        self.schedule = pd.read_csv(schedule_file)
        for course in self.schedule['course'].unique():
            self.attendance_record[course] = [0, 0]

    def record_attendance(self, course, attended):
        if course in self.attendance_record:
            self.attendance_record[course][0] += 1 if attended else 0
            self.attendance_record[course][1] += 1

    def check_class(self, current_time):
        current_day = current_time.strftime('%A')
        current_hour = current_time.hour
```

```
        upcoming_classes = self.schedule[
            (self.schedule['day'] == current_day) &
            (self.schedule['hour'] == current_hour)
        ]

        if not upcoming_classes.empty:
            class_info = upcoming_classes.iloc[0]
            course = class_info['course']
            attended, total = self.attendance_record[course]
            current_percentage = (attended / total * 100) if total > 0 else
100

            if current_percentage < self.attendance_goal:
                return f"URGENT: You must attend {course} in
{class_info['room']} to maintain {self.attendance_goal}% attendance. Current:
{current_percentage:.1f}%"
            return f"NOTIFICATION: You have {course} class in
{class_info['room']}. Attendance: {current_percentage:.1f}%"
        return "No classes scheduled at this time"
```

Testing Results:

• Goal Tracking: Successfully monitors progress toward attendance goals

• Accurate Calculations: Provides reliable attendance percentage calculations

• Effective Recommendations: Generates appropriate attendance-based notifications

```
Testing Goal-based Agent:
NOTIFICATION: You have Data Structures class in Hall-A101. Attendance: 100.0%
```

4. **Utility-Based Agent: Topic Priority System**

Objective: Develop an intelligent notification system that optimizes class attendance based on topic importance.

Approach and Logic:

• Multi-factor Analysis: Considers topic importance and course priority

• Utility Calculation: Computes importance scores based on topic significance

• Priority-based Optimization: Adjusts notifications based on topic importance

• Cost-benefit Analysis: Balances multiple factors to determine notification urgency

```
class UtilityBasedScheduleAgent:
    def __init__(self):
        self.schedule = None
        self.topic_importance = {}

    def load_schedule(self, schedule_file):
        self.schedule = pd.read_csv(schedule_file)

    def set_topic_importance(self, course, topic, importance):
        if course not in self.topic_importance:
            self.topic_importance[course] = {}
        self.topic_importance[course][topic] = importance

    def calculate_utility(self, course, topic):
        base_utility = 5
        topic_utility = self.topic_importance.get(course, {}).get(topic, 0)
        return base_utility + topic_utility

    def check_class(self, current_time):
        current_day = current_time.strftime('%A')
        current_hour = current_time.hour

        upcoming_classes = self.schedule[
            (self.schedule['day'] == current_day) &
            (self.schedule['hour'] == current_hour)
        ]

        if not upcoming_classes.empty:
            class_info = upcoming_classes.iloc[0]
            course = class_info['course']
            topic = class_info['topic']
            utility = self.calculate_utility(course, topic)

            if utility > 10:
                return f"CRITICAL: High-priority topic '{topic}' in {course}!
Must attend in {class_info['room']}!"
            elif utility > 7:
                return f"IMPORTANT: Valuable topic '{topic}' in {course} today
in {class_info['room']}."
            else:
                return f"NOTIFICATION: {course} class now in
{class_info['room']}. Topic: {topic}"
        return "No classes scheduled at this time"
```

Testing Results:

- Optimal Prioritization: Successfully identifies high-priority topics

• Importance Recognition: Effectively communicates topic significance

• User Guidance: Maintains appropriate notification levels based on topic importance

```
Testing Utility-based Agent:
CRITICAL: High-priority topic 'Binary Trees' in Data Structures! Must attend in Hall-A101!
```

5. **Learning Agent: Attendance Pattern System**

Objective: Create an adaptive system that learns and adjusts to student attendance patterns over time.

Approach and Logic:

• Pattern Recognition: Identifies patterns in class attendance

• Adaptive Learning: Gradually adjusts internal models based on attendance history

• Behaviour Tracking: Maintains historical data of attendance patterns

• Continuous Improvement: Regularly updates recommendations based on learned patterns

```
class LearningScheduleAgent:
    def __init__(self):
        self.schedule = None
        self.attendance_patterns = {}
        self.learning_rate = 0.1

    def load_schedule(self, schedule_file):
        self.schedule = pd.read_csv(schedule_file)

    def record_attendance_pattern(self, day, hour, course, attended):
        key = (day, hour, course)
        if key not in self.attendance_patterns:
            self.attendance_patterns[key] = 0.5

        current_prob = self.attendance_patterns[key]
        if attended:
            self.attendance_patterns[key] = current_prob + self.learning_rate
* (1 - current_prob)
        else:
            self.attendance_patterns[key] = current_prob + self.learning_rate
* (0 - current_prob)

    def check_class(self, current_time):
        current_day = current_time.strftime('%A')
```

```
        current_hour = current_time.hour

        upcoming_classes = self.schedule[
            (self.schedule['day'] == current_day) &
            (self.schedule['hour'] == current_hour)
        ]

        if not upcoming_classes.empty:
            class_info = upcoming_classes.iloc[0]
            course = class_info['course']
            key = (current_day, current_hour, course)
            attendance_prob = self.attendance_patterns.get(key, 0.5)

            if attendance_prob < 0.3:
                return f"ALERT: You frequently miss {course} on
{current_day}s! Please attend in {class_info['room']}!"
            elif attendance_prob < 0.6:
                return f"REMINDER: Your attendance in {course} on
{current_day}s could improve. Class in {class_info['room']}."
            else:
                return f"NOTIFICATION: You have {course} now in
{class_info['room']} - keep up your good attendance!"
        return "No classes scheduled at this time"
```

Testing Results:

• Learning Capability: Successfully adapts to attendance patterns over time

• Pattern Detection: Accurately identifies significant attendance trends

• Personalization: Provides increasingly personalized notifications with more data

```
Testing Learning Agent:
REMINDER: Your attendance in Data Structures on Mondays could improve. Class in Hall-A101.
```

Integration and System Architecture:

The implementation demonstrates how different types of agents can work together in a student planning system:

• Basic Schedule Management provides fundamental time-based notifications

• Holiday Awareness adds context-sensitive notifications

• Attendance Tracking ensures educational goals are met

• Topic Prioritization optimizes learning outcomes

• Pattern Learning provides personalized assistance

Each agent type builds upon the capabilities of the previous one, creating a comprehensive student planning system that combines reactive responses, model-based reasoning, goal-oriented behavior, utility optimization, and learning capabilities.