AI Partial 2

Erick Gonzalez Parada | 178145

1. Project specific topic:

Mi topico sera, crear una red neuronal que determine correctamente numeros nones y numeros pares.

2. Gradient Descent Algorithm

Implementation for finding the minimum of f(x) = x² + 4x + 5

running in collab: https://colab.research.google.com/drive/1aFuINJ2fXe5uj69SFi_JSz3X_ng9u7Jj?usp=sharing

```
import numpy as np
import matplotlib.pyplot as plt

def gradientDescent():
    """
    Implementation of gradient descent algorithm to find the minimum of
    f(x) = x^2 + 4x + 5

    Returns:
        tuple: List of x values, list of function values at each step
    """
    # Define the function f(x) = x^2 + 4x + 5
    def f(x):
        """
        Calculate the value of the function f(x) = x^2 + 4x + 5

        Args:
            x (float): Input value

        Returns:
            float: Function value at x
        """
        return x**2 + 4*x + 5

    # Define the derivative of f(x): f'(x) = 2x + 4
```

```
def fPrime(x):
    """
    Calculate the derivative of the function f'(x) = 2x + 4

    Args:
        x (float): Input value

    Returns:
        float: Derivative value at x
    """
    return 2*x + 4

# Initialize parameters
x0 = 1.0        # Initial starting point
alpha = 0.1     # Learning rate
iterations = 20 # Number of iterations

# Lists to store values for plotting
xValues = [x0]
fValues = [f(x0)]

# Current x value
x = x0

# Perform gradient descent iterations
for i in range(iterations):
    # Calculate gradient (derivative)
    gradient = fPrime(x)

    # Update x using gradient descent formula: x = x - alpha * gradient
    x = x - alpha * gradient

    # Store the new values
    xValues.append(x)
    fValues.append(f(x))

    # Print progress
    print(f"Iteration {i+1}: x = {x:.6f}, f(x) = {f(x):.6f}, gradient = {gradient:.6f}")

# Calculate the analytical minimum for verification
# For f(x) = x^2 + 4x + 5, the minimum is at x = -b/(2a) = -4/(2*1) = -2
analyticalMinimum = -2.0

print(f"\nAnalytical minimum: x = {analyticalMinimum}, f(x) = {f(analyticalMinimum)}")
```

```
    print(f"Final result after {iterations} iterations: x = {x:.6f}, f(x) = {f(x):.6f}")

    # Create visualization to show the convergence
    plotGradientDescent(f, xValues, fValues)

    return xValues, fValues

def plotGradientDescent(f, xValues, fValues):
    """
    Plot the function and the points obtained in each iteration

    Args:
        f (function): The function to minimize
        xValues (list): List of x values at each iteration
        fValues (list): List of function values at each iteration
    """
    # Create a range of x values for plotting the function
    xRange = np.linspace(-4, 2, 1000)
    yRange = [f(x) for x in xRange]

    # Create figure and axis
    plt.figure(figsize=(10, 6))

    # Plot the function
    plt.plot(xRange, yRange, 'b-', label='f(x) = x^2 + 4x + 5')

    # Plot the points from each iteration
    plt.plot(xValues[:-1], fValues[:-1], 'ro-', label='Gradient Descent Path')

    # Highlight the final point
    plt.plot(xValues[-1], fValues[-1], 'go', markersize=10, label='Final Point')

    # Mark the analytical minimum at x = -2
    analyticalMinimum = -2.0
    plt.plot(analyticalMinimum, f(analyticalMinimum), 'mo', markersize=10, label='Analytical Minim
um')

    # Add labels and title
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title('Gradient Descent Optimization for f(x) = x^2 + 4x + 5')
    plt.grid(True)
    plt.legend()
```

```
    # Show the plot
    plt.show()

# Execute the gradient descent algorithm
if __name__ == "__main__":
    xValues, fValues = gradientDescent()
```

Possible errors or issues:

1. **Learning Rate Selection**: If the learning rate (`alpha`) is too large, the algorithm might overshoot the minimum and diverge. If it's too small, convergence will be very slow.
2. **Initial Value Dependency**: The starting point (`x0`) can affect how quickly the algorithm converges, especially for more complex functions with multiple local minima.
3. **Precision Issues**: For very flat functions near the minimum, numerical precision might affect the accuracy of the final result.
4. **Termination Criteria**: Our implementation uses a fixed number of iterations, which might not be optimal. A better approach would be to stop when the change in x or function value is below a threshold.

3. Knowledge-Based System for Legal Case

running in collab: https://colab.research.google.com/drive/1aFuINJ2fXe5uj69SFi_JSz3X_ng9u7Jj?usp=sharing

Implementation of a system to determine guilt/innocence based on evidence

```
class KnowledgeBasedSystem:
    """
    A simple knowledge-based system using propositional logic to determine
    guilt or innocence in a legal case.

    This system uses a rule-based approach where facts are stored as boolean values
    and rules operate on these facts to determine conclusions.
    """


    def __init__(self):
        """
        Initialize the knowledge base with empty facts and rules
        """
        # Initialize facts and rules dictionaries
        self.facts = {}
        self.rules = []
```

```
        # Initialize the verdict
        self.verdict = None

        # Track which rules have been applied to prevent infinite loops
        self.appliedRules = set()

    def addFact(self, factName, value):
        """
        Add or update a fact in the knowledge base

        Args:
            factName (str): Name of the fact
            value (bool): Truth value of the fact
        """
        self.facts[factName] = value
        print(f"Added fact: {factName} = {value}")

    def addRule(self, conditions, conclusion, description):
        """
        Add a rule to the knowledge base

        Args:
            conditions (list): List of tuples (factName, expectedValue)
            conclusion (tuple): (factName, value) to be set if rule conditions are met
            description (str): Human-readable description of the rule
        """
        self.rules.append({
            'conditions': conditions,
            'conclusion': conclusion,
            'description': description
        })
        print(f"Added rule: {description}")

    def evaluateRule(self, rule):
        """
        Evaluate if a rule's conditions are met

        Args:
            rule (dict): Rule to evaluate

        Returns:
            bool: True if all conditions are met, False otherwise
        """
```

```
        # Check each condition in the rule
        for factName, expectedValue in rule['conditions']:
            # If fact doesn't exist or doesn't match expected value, rule doesn't apply
            if factName not in self.facts or self.facts[factName] != expectedValue:
                return False

        # If we get here, all conditions are met
        return True

    def applyRules(self):
        """
        Apply all rules to the current facts and update knowledge base

        Returns:
            bool: True if any rule was applied, False otherwise
        """
        ruleApplied = False

        # Check each rule
        for i, rule in enumerate(self.rules):
            # Create a unique identifier for this rule and its current application context
            factName, value = rule['conclusion']
            ruleId = f"{i}:{factName}:{value}"

            # Skip if this rule with this conclusion has already been applied
            if ruleId in self.appliedRules:
                continue

            if self.evaluateRule(rule):
                # Apply the conclusion of the rule
                factName, value = rule['conclusion']

                # Only apply the rule if it would change a fact
                if factName not in self.facts or self.facts[factName] != value:
                    self.facts[factName] = value
                    print(f"Applied rule: {rule['description']}")
                    print(f"Set {factName} = {value}")
                    ruleApplied = True

                    # Mark this rule as applied to prevent infinite loops
                    self.appliedRules.add(ruleId)

        return ruleApplied
```

```python
def inferenceEngine(self):
    """
    Run the inference engine until no more rules can be applied
    """
    print("\nStarting inference engine...")

    # Reset the applied rules tracking for a new inference run
    self.appliedRules = set()

    # Keep applying rules until no more can be applied
    iterationCount = 0
    maxIterations = 100  # Safety limit to prevent infinite loops

    while self.applyRules() and iterationCount < maxIterations:
        iterationCount += 1

    if iterationCount >= maxIterations:
        print("Warning: Reached maximum iterations. There might be a rule cycle.")

    print("Inference completed.")

def determineVerdict(self):
    """
    Determine the verdict based on the final state of the knowledge base
    """
    # Check if guilty fact exists and is true
    if 'is_guilty' in self.facts:
        self.verdict = "Guilty" if self.facts['is_guilty'] else "Innocent"
        print(f"\nVerdict: {self.verdict}")
    else:
        print("\nUnable to determine verdict with available facts.")

def resetCase(self):
    """
    Reset the case by clearing all facts but keeping the rules
    """
    self.facts = {}
    self.verdict = None
    self.appliedRules = set()
    print("Case has been reset. Facts cleared but rules retained.")

def appealCase(self, newFacts):
    """
    Appeal the case by adding new evidence (facts)
```

```
        Args:
            newFacts (dict): Dictionary of new facts (factName → value)
        """
        print("\nProcessing appeal with new evidence...")

        # Add new facts
        for factName, value in newFacts.items():
            self.addFact(factName, value)

        # Re-run inference
        self.inferenceEngine()

        # Determine new verdict
        self.determineVerdict()

# Example usage for all three cases
def runLegalCases():
    """
    Implement all three legal cases using the knowledge-based system
    """
    # Case 1: The Mansion Murder
    print("\n==================================")
    print("      THE MANSION MURDER CASE    ")
    print("==================================")

    # Create the knowledge-based system
    kbs = KnowledgeBasedSystem()

    # Define the rules
    print("\nDefining rules...")

    # Rule 1: If all incriminating evidence is present, butler is guilty
    kbs.addRule(
        conditions=[
            ('butler_near_scene', True),
            ('knife_has_fingerprints', True),
            ('butler_had_debt', True)
        ],
        conclusion=('is_guilty', True),
        description="If butler was near the scene, knife has fingerprints, and butler had debt, then but
    )

    # Rule 2: If security video exonerates butler, he wasn't at the scene
```

```
kbs.addRule(
    conditions=[('security_video_exonerates', True)],
    conclusion=('butler_near_scene', False),
    description="If security video shows butler elsewhere, then butler wasn't at the scene"
)

# Rule 3: If fingerprints don't match, knife evidence is invalid
kbs.addRule(
    conditions=[('fingerprints_match', False)],
    conclusion=('knife_has_fingerprints', False),
    description="If fingerprints don't match butler's, then knife evidence is invalid"
)

# Rule 4: If key evidence is missing, butler is innocent
kbs.addRule(
    conditions=[
        ('butler_near_scene', False),
        ('knife_has_fingerprints', False),
        ('butler_had_debt', True)
    ],
    conclusion=('is_guilty', False),
    description="If butler wasn't at scene and knife evidence is invalid, despite having debt, butler
)

# Initial facts
print("\nInitial case facts:")
kbs.addFact('butler_near_scene', True)
kbs.addFact('knife_has_fingerprints', True)
kbs.addFact('butler_had_debt', True)
kbs.addFact('security_video_exonerates', False)
kbs.addFact('fingerprints_match', True)

# Run inference engine
kbs.inferenceEngine()

# Determine initial verdict
kbs.determineVerdict()

# Process appeal with new evidence
print("\n======== APPEAL PROCESS ========")

# New facts for appeal
appealFacts = {
    'security_video_exonerates': True,  # Security video shows butler elsewhere
```

```
    'fingerprints_match': False         # Fingerprints don't match
}

# Appeal the case
kbs.appealCase(appealFacts)

# Case 2: The Bank Heist
print("\n==================================")
print("       THE BANK HEIST CASE        ")
print("==================================")

# Reset for new case
kbs.resetCase()

# Define the rules
print("\nDefining rules...")

# Rule 1: If all incriminating evidence is present, defendant is guilty
kbs.addRule(
    conditions=[
        ('had_access_to_blueprints', True),
        ('witness_saw_defendant', True),
        ('stolen_money_found', True)
    ],
    conclusion=('is_guilty', True),
    description="If defendant had access to blueprints, was seen at the scene, and stolen money
)

# Rule 2: If witness testimony is unreliable, it should not be considered
kbs.addRule(
    conditions=[('witness_testimony_reliable', False)],
    conclusion=('witness_saw_defendant', False),
    description="If witness testimony is unreliable, then it cannot be used as evidence"
)

# Rule 3: If money has legitimate source, it's not evidence of theft
kbs.addRule(
    conditions=[('money_has_legitimate_source', True)],
    conclusion=('stolen_money_found', False),
    description="If money has a legitimate source, then it's not evidence of theft"
)

# Rule 4: If key evidence is missing, defendant is innocent
kbs.addRule(
```

```
    conditions=[
        ('had_access_to_blueprints', True),
        ('witness_saw_defendant', False),
        ('stolen_money_found', False)
    ],
    conclusion=('is_guilty', False),
    description="If defendant had access to blueprints but wasn't seen at the scene and no stolen
)

# Initial facts (original case)
print("\nInitial case facts:")
kbs.addFact('had_access_to_blueprints', True)
kbs.addFact('witness_saw_defendant', True)
kbs.addFact('stolen_money_found', True)
kbs.addFact('witness_testimony_reliable', True)
kbs.addFact('money_has_legitimate_source', False)

# Run inference engine
kbs.inferenceEngine()

# Determine initial verdict
kbs.determineVerdict()

# Process appeal with new evidence
print("\n======== APPEAL PROCESS ========")

# New facts for appeal
appealFacts = {
    'witness_testimony_reliable': False,  # Witness admits they were mistaken
    'money_has_legitimate_source': True   # Money came from inheritance
}

# Appeal the case
kbs.appealCase(appealFacts)

# Case 3: The Traffic Accident
print("\n=================================")
print("     THE TRAFFIC ACCIDENT CASE    ")
print("=================================")

# Reset for new case
kbs.resetCase()

# Define the rules
```

```
print("\nDefining rules...")

# Rule 1: If all incriminating evidence is present, driver is guilty
kbs.addRule(
    conditions=[
        ('driver_was_speeding', True),
        ('driver_ran_red_light', True),
        ('driver_blood_alcohol_illegal', True)
    ],
    conclusion=('is_guilty', True),
    description="If driver was speeding, ran a red light, and had illegal blood alcohol, then driver is
)

# Rule 2: If traffic light analysis contradicts witness, light wasn't red
kbs.addRule(
    conditions=[('traffic_light_was_green', True)],
    conclusion=('driver_ran_red_light', False),
    description="If traffic light analysis shows green light, then driver didn't run a red light"
)

# Rule 3: If blood alcohol was legal, that evidence is invalid
kbs.addRule(
    conditions=[('blood_alcohol_within_limit', True)],
    conclusion=('driver_blood_alcohol_illegal', False),
    description="If blood alcohol was within legal limit, then driver wasn't illegally intoxicated"
)

# Rule 4: If key evidence is missing, driver is innocent
kbs.addRule(
    conditions=[
        ('driver_was_speeding', True),
        ('driver_ran_red_light', False),
        ('driver_blood_alcohol_illegal', False)
    ],
    conclusion=('is_guilty', False),
    description="If driver was speeding but didn't run a red light and wasn't illegally intoxicated, th
)

# Initial facts
print("\nInitial case facts:")
kbs.addFact('driver_was_speeding', True)
kbs.addFact('driver_ran_red_light', True)
kbs.addFact('driver_blood_alcohol_illegal', True)
kbs.addFact('traffic_light_was_green', False)
```

```
    kbs.addFact('blood_alcohol_within_limit', False)

    # Run inference engine
    kbs.inferenceEngine()

    # Determine initial verdict
    kbs.determineVerdict()

    # Process appeal with new evidence
    print("\n======== APPEAL PROCESS ========")

    # New facts for appeal
    appealFacts = {
        'traffic_light_was_green': True,     # Traffic light analysis shows green
        'blood_alcohol_within_limit': True   # Blood alcohol was within legal limit
    }

    # Appeal the case
    kbs.appealCase(appealFacts)

    return kbs

# Execute the legal cases
if __name__ == "__main__":
    legalSystem = runLegalCases()
```

Explanation of Formalism Used

I implemented this system using **propositional logic** with a rule-based approach for the following reasons:

1. **Simplicity and Clarity**: Propositional logic provides a straightforward way to represent facts (true/false statements) and rules (if-then relationships), making the system easy to understand.
2. **Natural Fit for Legal Reasoning**: Legal cases often involve discrete facts and clear rules of inference, which map well to propositional logic.
3. **Transparency**: The rule-based system allows for clear explanation of how a verdict was reached, which is crucial in legal systems.
4. **Flexibility for Appeals**: New evidence can easily be incorporated by updating the facts and re-running the inference engine.

Possible Issues or Limitations:

1. **Binary Nature**: Propositional logic only handles true/false values, whereas real legal reasoning often involves degrees of certainty or probability.

2. **Rule Ordering**: The system may be sensitive to the order in which rules are checked. A more sophisticated implementation might need to address this.
3. **Lack of Uncertainty Handling**: The system cannot represent partial beliefs or conflicting evidence well. A more advanced system might use fuzzy logic or probabilistic reasoning.
4. **Limited Expressiveness**: Complex relationships between facts may be difficult to express in propositional logic. First-order logic would provide more expressiveness but at the cost of complexity.
5. **Infinite Loops**: The original implementation could potentially fall into infinite loops if rules continuously trigger each other. I've fixed this by tracking applied rules and adding a maximum iteration limit.

4. Emotion Detection in Text (Bayesian Classifier)

running in collab: https://colab.research.google.com/drive/1aFuINJ2fXe5uj69SFi_JSz3X_ng9u7Jj?usp=sharing

Implementation of a system to detect emotions in text using Bayes' Theorem

```
import re
import math
from collections import defaultdict, Counter

class BayesianEmotionDetector:
    """
    A Bayesian system for detecting emotions in text messages.

    This system uses Bayes' Theorem to classify text into different emotion categories
    based on the words and their associated probabilities.
    """

    def __init__(self):
        """
        Initialize the emotion detector with empty training data
        """
        # Dictionary to store word frequencies for each emotion
        self.wordFrequencies = defaultdict(Counter)

        # Dictionary to store prior probabilities for each emotion
        self.priorProbabilities = {}

        # Set to store all unique words in the training data
```

```
        self.vocabulary = set()

        # Total number of messages by emotion
        self.emotionCounts = Counter()

        # Total number of messages
        self.totalMessages = 0

        # Smoothing parameter for Laplace smoothing
        self.alpha = 1.0

    def preprocessText(self, text):
        """
        Preprocess the text by converting to lowercase and removing punctuation

        Args:
            text (str): Text message to preprocess

        Returns:
            list: List of preprocessed words
        """
        # Convert to lowercase
        text = text.lower()

        # Remove punctuation and split into words
        words = re.findall(r'\b\w+\b', text)

        return words

    def train(self, trainingData):
        """
        Train the Bayesian model using labeled text data

        Args:
            trainingData (list): List of tuples (text, emotion)
        """
        print("Training Bayesian Emotion Detector...")

        # Reset training data
        self.wordFrequencies = defaultdict(Counter)
        self.emotionCounts = Counter()
        self.vocabulary = set()
        self.totalMessages = len(trainingData)
```

```
        # Process each training example
        for text, emotion in trainingData:
            # Preprocess the text
            words = self.preprocessText(text)

            # Update emotion counts
            self.emotionCounts[emotion] += 1

            # Update word frequencies for this emotion
            for word in words:
                self.wordFrequencies[emotion][word] += 1
                self.vocabulary.add(word)

        # Calculate prior probabilities
        for emotion in self.emotionCounts:
            self.priorProbabilities[emotion] = self.emotionCounts[emotion] / self.totalMessages

        print(f"Training completed with {self.totalMessages} messages and {len(self.vocabulary)} u
nique words.")
        print(f"Emotions distribution: {dict(self.emotionCounts)}")

    def calculateCondProb(self, word, emotion):
        """
        Calculate conditional probability P(word|emotion) with Laplace smoothing

        Args:
            word (str): The word
            emotion (str): The emotion

        Returns:
            float: Conditional probability P(word|emotion)
        """
        # Get count of the word for this emotion
        wordCount = self.wordFrequencies[emotion].get(word, 0)

        # Get total words for this emotion
        totalWordsForEmotion = sum(self.wordFrequencies[emotion].values())

        # Calculate P(word|emotion) with Laplace smoothing
        # (word_count + alpha) / (total_words + alpha * vocabulary_size)
        return (wordCount + self.alpha) / (totalWordsForEmotion + self.alpha * len(self.vocabulary))

    def predict(self, text):
        """
```

```
    Predict the emotion for a given text message using Bayes' Theorem

    Args:
        text (str): The text message

    Returns:
        tuple: (predicted_emotion, probabilities_dict)
    """
    # Preprocess the text
    words = self.preprocessText(text)

    # Dictionary to store probabilities for each emotion
    probabilities = {}

    # Calculate P(emotion|words) for each emotion
    for emotion in self.priorProbabilities:
        # Start with log of prior probability
        logProb = math.log(self.priorProbabilities[emotion])

        # Add log probabilities for each word
        for word in words:
            if word in self.vocabulary:
                condProb = self.calculateCondProb(word, emotion)
                logProb += math.log(condProb)

        # Store the log probability
        probabilities[emotion] = logProb

    # Find emotion with highest probability
    predictedEmotion = max(probabilities, key=probabilities.get)

    # Convert log probabilities to regular probabilities
    # First normalize to avoid underflow
    maxLogProb = max(probabilities.values())
    normalizedProbs = {e: math.exp(p - maxLogProb) for e, p in probabilities.items()}

    # Then normalize to get probabilities that sum to 1
    totalProb = sum(normalizedProbs.values())
    finalProbs = {e: p / totalProb for e, p in normalizedProbs.items()}

    return predictedEmotion, finalProbs

def explainPrediction(self, text, predictedEmotion, probabilities):
    """
```

```
        Explain the prediction by showing word contributions

        Args:
            text (str): The text message
            predictedEmotion (str): The predicted emotion
            probabilities (dict): Probabilities for each emotion
        """
        words = self.preprocessText(text)

        print(f"\nExplanation for prediction '{predictedEmotion}' (probability: {probabilities[predicte
dEmotion]:.4f}):")
        print(f"Text: '{text}'")
        print("\nWord contributions:")

        for word in words:
            if word in self.vocabulary:
                print(f"  - '{word}':")
                for emotion in self.priorProbabilities:
                    condProb = self.calculateCondProb(word, emotion)
                    print(f"    - P({word}|{emotion}) = {condProb:.4f}")

        print("\nPrior probabilities:")
        for emotion, prob in self.priorProbabilities.items():
            print(f"  - P({emotion}) = {prob:.4f}")

        print("\nFinal probabilities:")
        for emotion, prob in probabilities.items():
            print(f"  - P({emotion}|text) = {prob:.4f}")

# Example usage with sample data
def emotionDetectionDemo():
    """
    Demonstrate the Bayesian Emotion Detection system
    """
    print("================================")
    print("    EMOTION DETECTION DEMO     ")
    print("================================")

    # Create the emotion detector
    detector = BayesianEmotionDetector()

    # Example training data (text, emotion)
    trainingData = [
        ("I am so happy today", "happy"),
```

```
    ("What a great day", "happy"),
    ("Feeling joyful and excited", "happy"),
    ("This is wonderful news", "happy"),
    ("I got a promotion, I'm so happy", "happy"),

    ("I feel so sad today", "sad"),
    ("This is disappointing news", "sad"),
    ("I'm down and depressed", "sad"),
    ("My heart is broken", "sad"),
    ("I failed my exam, feeling sad", "sad"),

    ("I am so angry right now", "angry"),
    ("This makes me furious", "angry"),
    ("I hate when this happens", "angry"),
    ("I'm mad at you", "angry"),
    ("This is so frustrating", "angry")
]

# Train the model
detector.train(trainingData)

# Example test messages
testMessages = [
    "I'm so happy and joyful today",
    "This is such a sad story, I'm feeling down",
    "I'm angry and frustrated with the service",
    "Today is great but I'm feeling a bit sad",
    "This is frustrating me so much",
    "The news was good, I feel happy"
]

# Test the model
print("\nTesting the emotion detector:")
for message in testMessages:
    predictedEmotion, probabilities = detector.predict(message)
    print(f"\nMessage: '{message}'")
    print(f"Predicted Emotion: {predictedEmotion}")
    print("Probabilities:")
    for emotion, prob in probabilities.items():
        print(f"  - {emotion}: {prob:.4f}")

# Demonstrate handling unknown words
print("\n\nHandling unknown words:")
unknownMessage = "This is a completely unfamiliar message with strange vocabulary"
```

```
    predictedEmotion, probabilities = detector.predict(unknownMessage)
    print(f"\nMessage: '{unknownMessage}'")
    print(f"Predicted Emotion: {predictedEmotion}")
    print("Probabilities:")
    for emotion, prob in probabilities.items():
        print(f"  - {emotion}: {prob:.4f}")

    # Provide a detailed explanation for one prediction
    print("\n\nDetailed explanation of a prediction:")
    exampleMessage = "I'm feeling great and joyful today"
    predictedEmotion, probabilities = detector.predict(exampleMessage)
    detector.explainPrediction(exampleMessage, predictedEmotion, probabilities)

    return detector

# Execute the emotion detection demo
if __name__ == "__main__":
    emotionDetector = emotionDetectionDemo()
```

Possible Issues or limitations:

1. **Limited Vocabulary**: The system struggles with words not seen during training, leading to incorrect predictions for out-of-vocabulary words.
2. **Overfitting**: If the training data is small or not diverse, the model may overfit, reducing its ability to generalize to new text.
3. **Context Ignorance**: The system treats words independently, ignoring context or word order, which can lead to misinterpretation of phrases or sarcasm.
4. **Bias in Training Data**: Biased training data can skew predictions, especially for underrepresented emotions, reducing overall accuracy.

5. Fuzzy Logic

running in collab: https://colab.research.google.com/drive/1aFuINJ2fXe5uj69SFi_JSz3X_ng9u7Jj?usp=sharing

Implementation

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Define fuzzy variables
```

```
time_of_day = ctrl.Antecedent(np.arange(0, 24, 1), 'time_of_day')
occupancy = ctrl.Antecedent(np.arange(0, 2, 1), 'occupancy')
lighting_intensity = ctrl.Consequent(np.arange(0, 101, 1), 'lighting_intensity')

# Define membership functions for time_of_day
time_of_day['morning'] = fuzz.trimf(time_of_day.universe, [6, 9, 12])
time_of_day['afternoon'] = fuzz.trimf(time_of_day.universe, [12, 15, 18])
time_of_day['evening'] = fuzz.trimf(time_of_day.universe, [18, 21, 24])
time_of_day['night'] = fuzz.trimf(time_of_day.universe, [0, 3, 6])

# Define membership functions for occupancy
occupancy['unoccupied'] = fuzz.trimf(occupancy.universe, [0, 0, 1])
occupancy['occupied'] = fuzz.trimf(occupancy.universe, [0, 1, 1])

# Define membership functions for lighting_intensity
lighting_intensity['off'] = fuzz.trimf(lighting_intensity.universe, [0, 0, 25])
lighting_intensity['low'] = fuzz.trimf(lighting_intensity.universe, [0, 25, 50])
lighting_intensity['medium'] = fuzz.trimf(lighting_intensity.universe, [25, 50, 75])
lighting_intensity['high'] = fuzz.trimf(lighting_intensity.universe, [50, 75, 100])

# Define fuzzy rules
rule1 = ctrl.Rule(time_of_day['morning'] & occupancy['occupied'], lighting_intensity['medium'])
rule2 = ctrl.Rule(time_of_day['afternoon'] & occupancy['occupied'], lighting_intensity['low'])
rule3 = ctrl.Rule(time_of_day['evening'] & occupancy['occupied'], lighting_intensity['high'])
rule4 = ctrl.Rule(time_of_day['night'] & occupancy['unoccupied'], lighting_intensity['off'])
rule5 = ctrl.Rule(time_of_day['afternoon'] & occupancy['unoccupied'], lighting_intensity['off'])

# Create control system
lighting_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5])
lighting_system = ctrl.ControlSystemSimulation(lighting_ctrl)

# Example usage
def adjust_lighting(time, occupancy_status):
    lighting_system.input['time_of_day'] = time
    lighting_system.input['occupancy'] = occupancy_status
    lighting_system.compute()

        # Check if 'lighting_intensity' is in the output before accessing it
    if 'lighting_intensity' in lighting_system.output:
        return lighting_system.output['lighting_intensity']
    else:
        # Return a default value or handle the case when no rule is activated
        print("Warning: No rule activated for the given inputs. Returning 0 lighting intensity.")
        return 0  # Or any other appropriate default value
```

```
# Test the system
print("Lighting intensity at 7 AM, Occupied:", adjust_lighting(7, 1))
print("Lighting intensity at 1 PM, Unoccupied:", adjust_lighting(13, 0))
print("Lighting intensity at 7 PM, Occupied:", adjust_lighting(19, 1))
print("Lighting intensity at 11 PM, Unoccupied:", adjust_lighting(23, 0))
```

Possible errors or limitations in the Fuzzy Logic System

1. **Sensor Data Accuracy**: The system relies on accurate sensor data for time of day and occupancy. If the sensors provide incorrect or noisy data, the lighting control may not function as expected. For example, a malfunctioning motion sensor might incorrectly report a room as unoccupied.
2. **Membership Function Design**: The effectiveness of the fuzzy logic system depends on the proper design of membership functions. If the ranges for "morning," "afternoon," "evening," and "night" are not well-defined, the system might produce suboptimal lighting adjustments.
3. **Rule Base Completeness**: The current set of rules covers basic scenarios, but it may not handle edge cases or more complex situations. For example, what happens if the room is occupied during the night? Additional rules may be needed to cover all possible scenarios.
4. **User Preferences**: The system assumes fixed user preferences for lighting levels at different times of the day. However, user preferences can vary, and the system may need to allow for customization or learning from user behavior over time.