Data Science Assignment

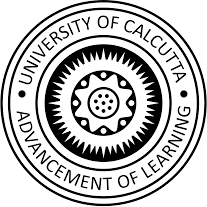
Semester 3

Paper: Data Science Laboratory (MCA P31)

Subject: Computer Application

Course: MCA (2 years)

University of Calcutta



|  |  |  |
| --- | --- | --- |
| Name | Registration No. | Roll Number |
| Deep Saha | 211-1111-0439-20 | C91/MCA/232005 |

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Serial No.** | **Topic** | **Page No.** | **Signature** |
| 1. | Genetic Algorithm | 1 - 6 |  |
| 2. | Ant Colony Optimization Algorithm | 7– 10 |  |
| 3. | Gradient Descent | 11 - 13 |  |
| 4. | K-Means Clustering and Gaussian Mixture Model | 14 – 16 |  |
| 5. | Adam Optimizer | 17 – 18 |  |
| 6 | Neural Network | 19-21 |  |
| 7 | Convolution in Image Processing | 22-27 |  |
| 8 | Fuit 360 Classification | 28 - 32 |  |
| 9 | Relationship Establishment between Two Variables | 33-37 |  |
| 10 | Classification of Good and Bad Customer | 38-43 |  |

#### Genetic Algorithm

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms.

Genetic algorithms simulate the process of natural selection which means those species that can adapt to changes in their environment can survive and reproduce and go to the next generation. In simple words, they simulate “survival of the fittest” among individuals of consecutive generations to solve a problem. Each generation consists of a population of individuals and each individual represents a point in search space and possible solution.

Following is the foundation of GAs based on this analogy –

1. Individuals in the population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others
3. Genes from the “fittest” parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.
4. Thus each successive generation is more suited for their environment.

##### Search space

The population of individuals are maintained within search space. Each individual represents a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).

##### Fitness Score

A Fitness Score is given to each individual which **shows the ability of an individual to “compete”**. The individual having optimal fitness score (or near optimal) are sought.

The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores.The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce **better offspring** by combining chromosomes of parents. The population size is static so the room has to be created for new arrivals. So, some individuals die and get

replaced by new arrivals eventually creating new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.

Each new generation has on average more “better genes” than the individual (solution) of previous generations. Thus each new generations have better **“partial solutions”** than previous generations.

##### Operators of Genetic Algorithms

Once the initial generation is created, the algorithm evolves the generation using following operators –

1. **Selection Operator:** The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.
2. **Crossover Operator:** This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring).
3. **Mutation Operator:** The key idea is to insert random genes in offspring to maintain the diversity in the population to avoid premature convergence. For example –

##### The whole algorithm can be summarized as –

1. Randomly initialize populations p
2. Determine fitness of population
3. Until convergence repeat:
   1. Select parents from population
   2. Crossover and generate new population
   3. Perform mutation on new population
   4. Calculate fitness for new population

##### Code :

import random

distance\_matrix = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

def initialize\_population(pop\_size, n):

population = []

for \_ in range(pop\_size): individual = list(range(n)) random.shuffle(individual) population.append(individual)

return population

def fitness(path): total\_distance = 0

for i in range(len(path) - 1):

total\_distance += distance\_matrix[path[i]][path[i+1]]

total\_distance += distance\_matrix[path[-1]][path[0]]

return total\_distance

def tournament\_selection(population): tournament\_size = 3

selected = random.sample(population, tournament\_size)

selected.sort(key=fitness)

return selected[0]

def ordered\_crossover(parent1, parent2): size = len(parent1)

start, end = sorted(random.sample(range(size), 2))

child = [-1] \* size

child[start:end+1] = parent1[start:end+1]

current\_position = end + 1 for i in range(size):

if parent2[i] not in child:

|  |
| --- |
| if current\_position == size:  current\_position = 0 |
| child[current\_position] = parent2[i] current\_position += 1  return child |
| def swap\_mutation(path): |
| idx1, idx2 = random.sample(range(len(path)), 2)  path[idx1], path[idx2] = path[idx2], path[idx1] |
| def genetic\_algorithm(pop\_size, n\_generations, n\_cities): population = initialize\_population(pop\_size, n\_cities)  for generation in range(n\_generations): new\_population = []  for \_ in range(pop\_size // 2):  parent1 = tournament\_selection(population) parent2 = tournament\_selection(population)  child1 = ordered\_crossover(parent1, parent2) child2 = ordered\_crossover(parent2, parent1)  if random.random() < 0.1: swap\_mutation(child1)  if random.random() < 0.1: swap\_mutation(child2)  new\_population.extend([child1, child2]) population = new\_population  best\_individual = min(population, key=fitness)  print(f"Generation {generation+1}, Best Path: {best\_individual}, Best Distance:  {fitness(best\_individual)}")  best\_solution = min(population, key=fitness) return best\_solution, fitness(best\_solution)  pop\_size = 100 |

|  |  |
| --- | --- |
| n\_generations = 100  n\_cities = 4 | |
| best\_path, best\_distance = genetic\_algorithm(pop\_size, n\_generations, n\_cities) | |
| print("\nFinal Best Path:", best\_path) |  |
| print("Final Best Distance:", best\_distance) | |

##### Output :

Generation 1, Best Path: [2, 3, 1, 0], Best Distance: 80

Generation 2, Best Path: [2, 3, 1, 0], Best Distance: 80

Generation 3, Best Path: [3, 2, 0, 1], Best Distance: 80

**.**

**.**

**.**

Generation 99, Best Path: [0, 1, 3, 2], Best Distance: 80

Generation 100, Best Path: [0, 2, 3, 1], Best Distance: 80

Final Best Path: [0, 2, 3, 1] Final Best Distance: 80

#### Ant Colony Optimization Algorithm

Ants are eusocial insects that prefer community survival and sustaining rather than as ACO is an optimization algorithm inspired by ants' natural foraging behavior. Ants find the shortest paths by laying down pheromones, which guide other ants. Over time, stronger pheromone trails attract more ants, reinforcing shorter, better paths. ACO uses this process to solve problems like the Traveling Salesman Problem (TSP) by finding optimal or near-optimal solutions.

##### Steps :

1. **Initialize Parameters**: Set the number of ants, pheromone decay rate, and other parameters.
2. **Initialize Pheromones**: Start with small pheromone values on all paths.
3. **Ant Movement**: Each ant moves through the problem space, choosing paths based on pheromone strength and heuristic information.
4. **Pheromone Update**: After ants complete their paths, pheromones are updated. The best solutions receive stronger pheromones.
5. **Evaporation**: Pheromones decay over time to allow new paths to be explored.
6. **Termination**: Stop after a set number of iterations or when no significant improvement occurs.

##### Code :

import random import numpy as np

distance\_matrix = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

n\_cities = len(distance\_matrix) n\_ants = 10

n\_iterations = 100

alpha = 1 # pheromone importance

beta = 2 # heuristic (distance) importance rho = 0.1 # pheromone evaporation rate Q = 100 # pheromone deposit constant

pheromone = np.ones((n\_cities, n\_cities))

heuristic = 1.0 / np.array(distance\_matrix) np.fill\_diagonal(heuristic, 0)

def total\_distance(tour): dist = 0

for i in range(len(tour) - 1):

dist += distance\_matrix[tour[i]][tour[i + 1]]

dist += distance\_matrix[tour[-1]][tour[0]]

return dist

def choose\_next\_city(current\_city, visited\_cities): probabilities = []

for city in range(n\_cities):

if city not in visited\_cities:

pheromone\_value = pheromone[current\_city][city] \*\* alpha

heuristic\_value = heuristic[current\_city][city] \*\* beta probability = pheromone\_value \* heuristic\_value probabilities.append(probability)

else:

probabilities.append(0)

total = sum(probabilities)

probabilities = [p / total for p in probabilities]

return np.random.choice(range(n\_cities), p=probabilities)

def update\_pheromone(ants\_paths, ants\_distances):

global pheromone

pheromone \*= (1 - rho)

for i, path in enumerate(ants\_paths): for j in range(len(path) - 1):

pheromone[path[j]][path[j + 1]] += Q / ants\_distances[i]

pheromone[path[-1]][path[0]] += Q / ants\_distances[i] best\_path = None

best\_distance = float('inf')

for iteration in range(n\_iterations):

ants\_paths = []

ants\_distances = []

for ant in range(n\_ants):

visited\_cities = [random.randint(0, n\_cities - 1)] while len(visited\_cities) < n\_cities:

next\_city = choose\_next\_city(visited\_cities[-1], visited\_cities)

visited\_cities.append(next\_city)

ants\_paths.append(visited\_cities)

distance = total\_distance(visited\_cities) ants\_distances.append(distance)

print(f"Ant {ant + 1} Path: {visited\_cities}, Distance: {distance}")

min\_distance = min(ants\_distances)

if min\_distance < best\_distance: best\_distance = min\_distance

best\_path = ants\_paths[ants\_distances.index(min\_distance)]

update\_pheromone(ants\_paths, ants\_distances)

print(f"Iteration {iteration + 1}, Best Distance: {best\_distance}\n")

print("\nBest Path Found:", best\_path) print("Best Distance Found:", best\_distance)

##### Output :

Ant 1 Path: [0, 1, 2, 3], Distance: 95

Ant 2 Path: [3, 0, 1, 2], Distance: 95

Ant 3 Path: [0, 1, 3, 2], Distance: 80

Ant 4 Path: [0, 1, 3, 2], Distance: 80

Ant 5 Path: [3, 0, 1, 2], Distance: 95

Ant 6 Path: [0, 1, 3, 2], Distance: 80

Ant 7 Path: [1, 3, 0, 2], Distance: 95

Ant 8 Path: [3, 0, 1, 2], Distance: 95

Ant 9 Path: [3, 2, 0, 1], Distance: 80

Ant 10 Path: [2, 3, 0, 1], Distance: 95

Iteration 1, Best Distance: 80

.

.

.

Ant 1 Path: [1, 0, 2, 3], Distance: 80

Ant 2 Path: [1, 0, 2, 3], Distance: 80

Ant 3 Path: [2, 0, 1, 3], Distance: 80

Ant 4 Path: [0, 1, 3, 2], Distance: 80

Ant 5 Path: [2, 0, 1, 3], Distance: 80

Ant 6 Path: [2, 0, 1, 3], Distance: 80

Ant 7 Path: [3, 1, 0, 2], Distance: 80

Ant 8 Path: [3, 1, 0, 2], Distance: 80

Ant 9 Path: [2, 0, 1, 3], Distance: 80

Ant 10 Path: [1, 0, 2, 3], Distance: 80

Iteration 100, Best Distance: 80

Best Path Found: [0, 1, 3, 2] Best Distance Found: 80

#### Gradient Descent

Gradient Descent is an optimization algorithm used to minimize a function by iteratively moving towards the steepest descent, i.e., the direction in which the function decreases the most. It’s commonly used in machine learning and deep learning for training models by minimizing a **loss function** or **cost function**.

##### Steps:

1. **Start with an initial guess**: You begin with a random or pre-defined value (often called the "starting point") for the parameters you are trying to optimize (e.g., weights in machine learning models).
2. **Calculate the gradient**: The gradient is a vector of partial derivatives of the function you want to minimize. It points in the direction of the steepest ascent. To minimize the function, we move in the opposite direction of the gradient.
3. **Update the parameters**: The parameters are adjusted in the direction that reduces the value of the function. This is done by subtracting a fraction of the gradient from the current value of the parameters.
4. **Repeat**: The process is repeated until the function reaches a minimum (or stops changing significantly).

The basic update rule for gradient descent is:

**θ=θ−α**∇**J(θ)**

Where:

* + Θ is the parameter we are optimizing (e.g., weights in a machine learning model).
  + Α is the **learning rate**: A small positive scalar that controls how big a step we take in the direction of the gradient. A smaller α results in more gradual steps.
  + ∇J(θ) is the **gradient of the cost function** J(θ) at the current parameter value. The

gradient is the vector of partial derivatives, and it indicates the direction of the steepest increase.

##### Code :

import numpy as np

import matplotlib.pyplot as plt

def cost\_function(theta): return theta\*\*2 + 4\*theta + 4

def gradient(theta):

return 2\*theta + 4

def gradient\_descent(learning\_rate, initial\_theta, iterations): theta = initial\_theta

for \_ in range(iterations):

theta -= learning\_rate \* gradient(theta)

return theta

# Parameters learning\_rate = 0.1

initial\_theta = 10

iterations = 10

optimal\_theta = gradient\_descent(learning\_rate, initial\_theta, iterations) final\_cost = cost\_function(optimal\_theta)

print(f"Optimal Theta: {optimal\_theta}") print(f"Final Cost: {final\_cost}")

theta\_values = np.linspace(-10, 10, 100) cost\_values = cost\_function(theta\_values)

plt.plot(theta\_values, cost\_values, label="Cost Function $J(\\theta)$", color="blue") plt.scatter(optimal\_theta, final\_cost, color="red", label=f"Optimal $\\theta = {optimal\_theta:.2f}$") plt.xlabel('Theta')

plt.ylabel('Cost')

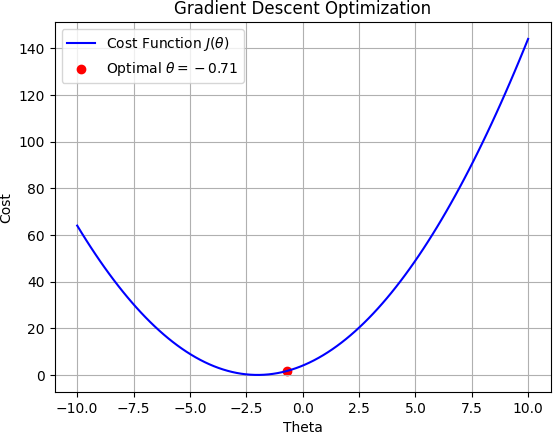
plt.title('Gradient Descent Optimization') plt.legend()

plt.grid() plt.show()

##### Output :

Optimal Theta: -0.7115098112000002

Final Cost: 1.6602069666338588



#### K-Means & Gaussian Mixture

##### K-Means Clustering

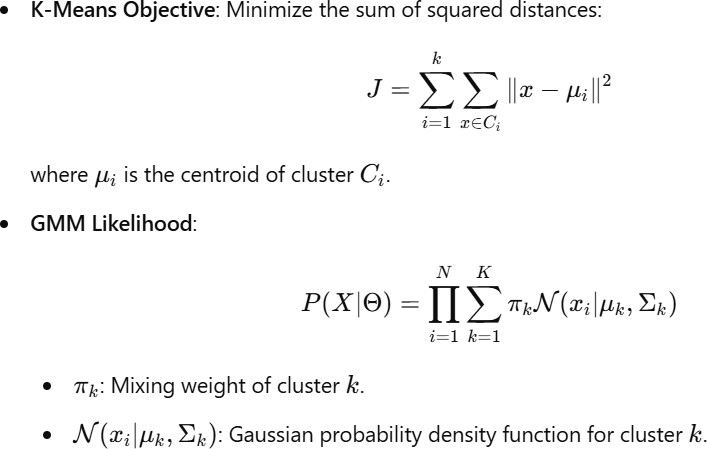
K-Means is an unsupervised learning algorithm used for clustering data into k groups.

1. Goal: Partition data into k clusters such that data points in the same cluster are as close as possible to each other (minimizing variance).
2. How It Works:
   * Initialize k centroids randomly.
   * Assign each data point to the nearest centroid.
   * Update the centroids as the mean of the data points assigned to them.
   * Repeat assign and update steps until the centroids stop changing or the change is minimal.
3. Limitations:
   * Assumes clusters are spherical and equally sized.
   * Sensitive to outliers.
   * Requires k to be predefined.

##### Gaussian Mixture Model (GMM)

A Gaussian Mixture Model is a probabilistic approach to clustering, assuming data is generated from a mixture of several Gaussian distributions.

1. Goal: Model the data as a combination of multiple Gaussian distributions, each representing a cluster.
2. How It Works:
   * Instead of assigning each point to one cluster (as in K-Means), GMM calculates the probability that each point belongs to a cluster.
   * Uses the Expectation-Maximization (EM) algorithm:
     + E-step: Compute the probabilities (responsibilities) that each point belongs to each Gaussian component.
     + M-step: Update the parameters of the Gaussian components (mean, variance, and mixing weights) to maximize the likelihood of the data.
   * Iterates until convergence.
3. Advantages:
   * Can model complex, non-spherical clusters.
   * Accounts for cluster overlap by using probabilities.
4. Comparison with K-Means:
   * K-Means assigns data points hard cluster labels; GMM assigns soft probabilistic labels.
   * GMM is more flexible and robust for non-linear, overlapping data distributions.



##### Code :

import numpy as np

from sklearn.datasets import load\_iris from sklearn.cluster import KMeans

from sklearn.mixture import GaussianMixture from sklearn.metrics import adjusted\_rand\_score

data = load\_iris()

X = data.data y = data.target

kmeans = KMeans(n\_clusters=3, random\_state=42) kmeans\_labels = kmeans.fit\_predict(X)

gmm = GaussianMixture(n\_components=3, random\_state=42) gmm\_labels = gmm.fit\_predict(X)

kmeans\_accuracy = adjusted\_rand\_score(y, kmeans\_labels) gmm\_accuracy = adjusted\_rand\_score(y, gmm\_labels)

print(f"K-Means Accuracy : {kmeans\_accuracy:.4f}") print(f"GMM Accuracy : {gmm\_accuracy:.4f}")

##### Output :

K-Means Accuracy : 0.7163 GMM Accuracy : 0.9039

#### Adam Optimizer

Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problem involving a lot of data or parameters. It requires less memory and is efficient. Intuitively, it is a combination of the ‘gradient descent with momentum’ algorithm and the ‘RMSP’ algorithm.

Adam optimizer involves a combination of two gradient descent methodologies:

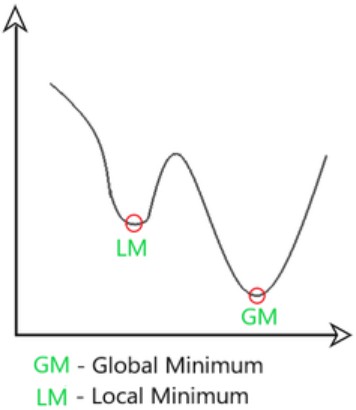
##### Momentum:

This algorithm is used to accelerate the gradient descent algorithm by taking into consideration the ‘exponentially weighted average’ of the gradients. Using averages makes the algorithm converge towards the minima in a faster pace.

##### Root Mean Square Propagation (RMSP):

Root mean square prop or RMSprop is an adaptive learning algorithm that tries to improve AdaGrad. Instead of taking the cumulative sum of squared gradients like in AdaGrad, it takes the ‘exponential moving average’.

Adam Optimizer inherits the strengths or the positive attributes of the above two methods and builds upon them to give a more optimized gradient descent.



##### Code :

import tensorflow as tf

from tensorflow.keras import Sequential

from tensorflow.keras.layers import Dense, Flatten from tensorflow.keras.datasets import mnist

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0 # Normalizing pixel values to [0, 1]

model = Sequential([

Flatten(input\_shape=(28, 28)), # Flatten 28x28 images into 1D array Dense(128, activation='relu'), # Hidden layer with 128 neurons Dense(10, activation='softmax') # Output layer for 10 classes

])

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=5, batch\_size=32)

test\_loss, test\_accuracy = model.evaluate(x\_test, y\_test) print(f"Test Accuracy: {test\_accuracy:.2f}")

##### Output :

Epoch 1/5

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **9s** 4ms/step - accuracy: 0.8805 - loss: 0.4261 Epoch 2/5

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **6s** 3ms/step - accuracy: 0.9648 - loss: 0.1222 Epoch 3/5

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **8s** 4ms/step - accuracy: 0.9782 - loss: 0.0746 Epoch 4/5

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **9s** 3ms/step - accuracy: 0.9831 - loss: 0.0562 Epoch 5/5

**1875/1875** ━━━━━━━━━━━━━━━━━━━━ **8s** 4ms/step - accuracy: 0.9868 - loss: 0.0431

**313/313** ━━━━━━━━━━━━━━━━━━━━ **1s** 2ms/step - accuracy: 0.9723 - loss: 0.0833 Test Accuracy: 0.98

#### Neural Network

Neural networks are machine learning models that mimic the complex functions of the human brain. These models consist of interconnected nodes or neurons that process data, learn patterns, and enable tasks such as pattern recognition and decision-making.

Neural networks are capable of learning and identifying patterns directly from data without pre-defined rules. These networks are built from several key components:

1. **Neurons**: The basic units that receive inputs, each neuron is governed by a threshold and an activation function.
2. **Connections**: Links between neurons that carry information, regulated by

weights and biases.

1. **Weights and Biases**: These parameters determine the strength and influence of connections.
2. **Propagation Functions**: Mechanisms that help process and transfer data across

layers of neurons.

1. **Learning Rule**: The method that adjusts weights and biases over time to improve accuracy.

##### Learning in neural networks follows a structured, three-stage process:

1. **Input Computation**: Data is fed into the network.
2. **Output Generation**: Based on the current parameters, the network generates an output.
3. **Iterative Refinement**: The network refines its output by adjusting weights and

biases, gradually improving its performance on diverse tasks.

**Code :**

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

data = load\_iris()

X = data.data y = data.target

X = X[:, :2]

scaler = StandardScaler() X\_scaled = scaler.fit\_transform(X)

nn = MLPClassifier(hidden\_layer\_sizes=(10, 10), max\_iter=1000, activation='logistic', random\_state=42)

nn.fit(X\_scaled, y)

y\_pred = nn.predict(X\_scaled)

accuracy = accuracy\_score(y, y\_pred)

precision = precision\_score(y, y\_pred, average='weighted') recall = recall\_score(y, y\_pred, average='weighted')

f1 = f1\_score(y, y\_pred, average='weighted')

print(f"Accuracy: {accuracy:.4f}") print(f"Precision (weighted): {precision:.4f}") print(f"Recall (weighted): {recall:.4f}") print(f"F1 Score (weighted): {f1:.4f}")

##### Output :

Accuracy: 0.8200

Precision (weighted): 0.8206

Recall (weighted): 0.8200 F1 Score (weighted): 0.8198

convolution.md 2024-12-20

# Convolution in Image Processing

### Topic of the Assignment

This assignment focuses on applying convolution operations to images using different kernels.

### Objective

The objective is to understand how convolution works in image processing and to apply various kernels to an image to observe the effects.

### Process

1. **Import necessary libraries**: Import the required libraries such as OpenCV for image processing, Matplotlib for displaying images, and NumPy for numerical operations.
2. **Define a function to perform convolution on an image with a given kernel**: Create a function conv

that takes an image and a kernel as input and performs convolution operation.

1. **Apply different kernels to the image and display the results using matplotlib**: Use various kernels to convolve the image and display the results to observe the effects of different kernels.

**import** cv2

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

### Define Convolution Function

The conv function reads an image, applies the given kernel to perform convolution, and returns the processed image.

**def conv**(i, kernel):

im = cv2.imread(i, 0) *# Read the image in grayscale*

img = im.copy()

(krow, kcol) = kernel.shape[:2] *# Get the kernel dimensions*

(row, col) = img.shape[:2] *# Get the image dimensions*

**for** i **in** range(0, row - krow + 1):

**for** j **in** range(0, col - kcol + 1):

temp = (kernel \* im[i:i + krow, j:j + kcol]).sum() *# Apply the kernel*

**if** temp < 0:

temp = 0

**if** temp > 255: temp = 255

im[i][j] = temp

im = cv2.cvtColor(im, cv2.COLOR\_BGR2RGB) *# Convert the image to RGB*

**return** im

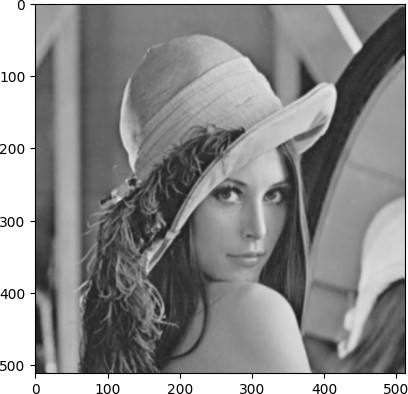
convolution.md 2024-12-20

### Apply Different Kernels

Apply various kernels to the image and display the results using Matplotlib.

Kernel 1: Averaging Filter

This kernel averages the pixel values in the neighborhood, resulting in a smoothing effect.



kernel = np.array([[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]]) new\_img = conv("lena.tiff", kernel)

plt.imshow(new\_img, cmap="gray")

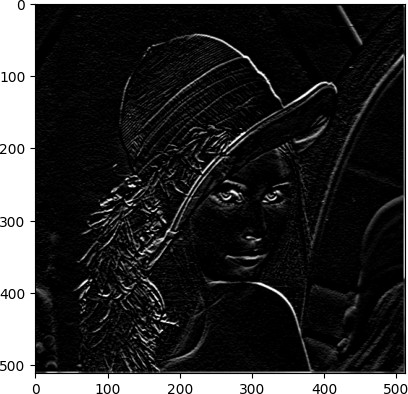
Kernel 2: Sobel Filter (Vertical Edge Detection)

This kernel detects vertical edges in the image.

kernel = np.array([[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]]) new\_img = conv("lena.tiff", kernel)

plt.imshow(new\_img, cmap="gray")

convolution.md 2024-12-20



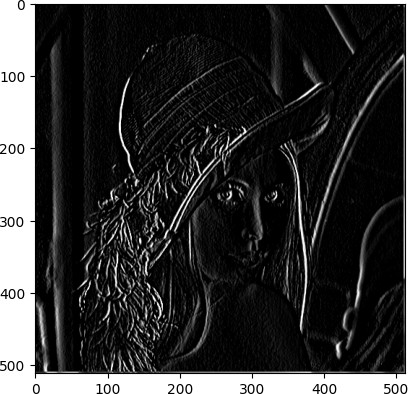
Kernel 3: Sobel Filter (Horizontal Edge Detection)

This kernel detects horizontal edges in the image.

kernel = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])

new\_img = conv("lena.tiff", kernel) plt.imshow(new\_img, cmap="gray")

convolution.md 2024-12-20



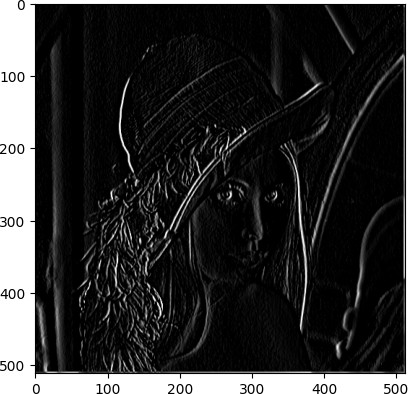
Kernel 4: Prewitt Filter (Vertical Edge Detection)

This kernel is another edge detection filter similar to Sobel but with different weights.

kernel = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])

new\_img = conv("lena.tiff", kernel) plt.imshow(new\_img, cmap="gray")

convolution.md 2024-12-20



fruits360.md 2024-12-20

# Fruit 360 Classification

### Topic of the Assignment

This assignment focuses on classifying different types of fruits using a custom dataset and a Convolutional Neural Network (CNN).

### Objective

The objective is to understand how to build and train a CNN model for image classification and to evaluate its performance on a test dataset.

### Process

1. **Import necessary libraries**: Import the required libraries such as PyTorch for building and training the model, OpenCV for image processing, and other utilities for data handling and evaluation.
2. **Define a custom dataset class**: Create a Fruits360Dataset class that loads images, applies transformations, and extracts features using SIFT and GLCM.
3. **Define the CNN model**: Create a CNNModel class that defines the architecture of the neural network.
4. **Prepare the data loaders**: Load the training and test datasets using the custom dataset class and prepare data loaders.
5. **Train the model**: Define a function to train the model and print the training loss.
6. **Evaluate the model**: Define a function to evaluate the model on the training and test datasets and print the accuracy, precision, and F1 score.

**import** torch

**import** torch.nn **as** nn

**import** torch.optim **as** optim

**import** torchvision.transforms **as** transforms

**from** torch.utils.data **import** DataLoader, Dataset

**from** sklearn.decomposition **import** PCA

**from** sklearn.metrics **import** accuracy\_score, precision\_score, f1\_score

**import** cv2

**import** numpy **as** np

**import** os

**from** skimage.feature **import** graycomatrix, graycoprops

**from** sklearn.preprocessing **import** StandardScaler

**from** torchvision **import** datasets

**import** random

### Define Custom Dataset Class

The Fruits360Dataset class loads images, applies transformations, and extracts features using SIFT and GLCM.

**class Fruits360Dataset**(Dataset):

**def**  **init** (self, data\_dir, transform=None, selected\_classes=None):

fruits360.md 2024-12-20

self.data\_dir = data\_dir

self.transform = transform

self.selected\_classes = selected\_classes self.image\_paths = []

self.labels = []

**for** i, class\_name **in** enumerate(self.selected\_classes): class\_dir = os.path.join(self.data\_dir, class\_name) **for** img\_name **in** os.listdir(class\_dir):

img\_path = os.path.join(class\_dir, img\_name) self.image\_paths.append(img\_path)

self.labels.append(i)

**def**  **len** (self):

**return** len(self.image\_paths)

**def**  **getitem** (self, idx):

img\_path = self.image\_paths[idx] image = cv2.imread(img\_path)

image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY) label = self.labels[idx]

image = cv2.resize(image, (100, 100))

**if** self.transform:

image = self.transform(image)

sift\_features = self.extract\_sift(image) glcm\_features = self.extract\_glcm(image)

features = np.concatenate((sift\_features, glcm\_features), axis=0)

**return** torch.tensor(features, dtype=torch.float32), label

**def extract\_sift**(self, image): sift = cv2.SIFT\_create()

keypoints, descriptors = sift.detectAndCompute(image, None)

**if** descriptors **is** None:

**return** np.zeros((128,))

**return** np.mean(descriptors, axis=0)

**def extract\_glcm**(self, image):

glcm = graycomatrix(image, distances=[1], angles=[0], levels=256, symmetric=True, normed=True)

contrast = graycoprops(glcm, 'contrast')[0, 0]

homogeneity = graycoprops(glcm, 'homogeneity')[0, 0]

**return** np.array([contrast, homogeneity])

**def pca\_transform**(self, image):

flat\_img = image.flatten().reshape(1, -1) pca = PCA(n\_components=20)

**return** pca.fit\_transform(flat\_img).flatten()

fruits360.md 2024-12-20

### Define CNN Model

The CNNModel class defines the architecture of the neural network.

**class CNNModel**(nn.Module):

**def**  **init** (self, input\_size, num\_classes): super(CNNModel, self). init ()

self.fc1 = nn.Linear(input\_size, 512) self.fc2 = nn.Linear(512, 256)

self.fc3 = nn.Linear(256, 128) self.fc4 = nn.Linear(128, 64)

self.fc5 = nn.Linear(64, num\_classes)

**def forward**(self, x):

x = torch.relu(self.fc1(x)) x = torch.relu(self.fc2(x)) x = torch.relu(self.fc3(x)) x = torch.relu(self.fc4(x)) x = self.fc5(x)

**return** x

### Prepare Data Loaders

Load the training and test datasets using the custom dataset class and prepare data loaders.

data\_dir = "./fruits"

test\_dir = "./test\_fruits"

selected\_classes = ["Banana 1", "Banana Lady Finger 1", "Banana Red 1"] transform = transforms.Compose([transforms.ToTensor()])

dataset = Fruits360Dataset(data\_dir, transform=None, selected\_classes=selected\_classes)

test\_dataset = Fruits360Dataset(test\_dir, transform=None, selected\_classes=selected\_classes)

train\_loader = DataLoader(dataset, batch\_size=32, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=32, shuffle=True)

input\_size = 128 + 2 *# SIFT(128), GLCM(2)*

num\_classes = len(selected\_classes)

model = CNNModel(input\_size=input\_size, num\_classes=num\_classes)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

### Train the Model

Define a function to train the model and print the training loss.

fruits360.md 2024-12-20

**def train\_model**(model, train\_loader, num\_epochs=10):

**for** epoch **in** range(num\_epochs): model.train()

running\_loss = 0.0

**for** inputs, labels **in** train\_loader: optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, labels) loss.backward()

optimizer.step()

running\_loss += loss.item()

print(f'Epoch {epoch+1}/{num\_epochs}, Loss:

{running\_loss/len(train\_loader)}')

|  |  |  |  |
| --- | --- | --- | --- |
| Epoch | 1/10, | Loss: | 1.0986123085021973 |
| Epoch | 2/10, | Loss: | 1.0986123085021973 |
| Epoch | 3/10, | Loss: | 1.0986123085021973 |
| Epoch | 4/10, | Loss: | 1.0986123085021973 |
| Epoch | 5/10, | Loss: | 1.0986123085021973 |
| Epoch | 6/10, | Loss: | 1.0986123085021973 |
| Epoch | 7/10, | Loss: | 1.0986123085021973 |
| Epoch | 8/10, | Loss: | 1.0986123085021973 |
| Epoch | 9/10, | Loss: | 1.0986123085021973 |

### Evaluate the Model

Epoch 10/10, Loss: 1.0986123085021973

Define a function to evaluate the model on the training and test datasets and print the accuracy, precision, and F1 score.

**def evaluate\_model**(model, train\_loader): model.eval()

y\_true = [] y\_pred = []

**with** torch.no\_grad():

**for** inputs, labels **in** train\_loader: outputs = model(inputs)

\_, predicted = torch.max(outputs, 1) y\_true.extend(labels.numpy())

y\_pred.extend(predicted.numpy())

accuracy = accuracy\_score(y\_true, y\_pred)

precision = precision\_score(y\_true, y\_pred, average='macro') f1 = f1\_score(y\_true, y\_pred, average='macro')

print(f'Accuracy: {accuracy}') print(f'Precision: {precision}') print(f'F1 Score: {f1}')

fruits360.md 2024-12-20

Accuracy: 0.3333333333333333

Precision: 0.3333333333333333

F1 Score: 0.3333333333333333

### Train and Evaluate the Model

Train the model and evaluate its performance on the training and test datasets.

train\_model(model, train\_loader, num\_epochs=10) evaluate\_model(model, train\_loader)

### Evaluate on Test Data

Evaluate the model on the test dataset and print the results.

print("Test score : ")

evaluate\_model(model, test\_loader)

Test score :

Accuracy: 0.3333333333333333

Precision: 0.3333333333333333

F1 Score: 0.3333333333333333

## Question 1:

1. Draw the strength and direction of the relationship between A and B
2. Converse the relationship between dependent and independent variable
3. Draw the best fit line using given dataset
4. Given a dataset with input features and target values , draw the best fit line that minimizes the difference between the predicted and actual values.

|  |  |
| --- | --- |
| A | B |
| 6.25 | 4.03 |
| 6.5 | 4.02 |
| 6.5 | 4.02 |
| 6 | 4.04 |
| 6.25 | 4.03 |
| 6.25 | 4.03 |

## Code:

import numpy as np import pandas as pd

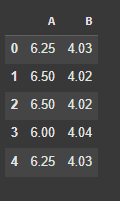
a=[6.25,6.5,6.5,6,6.25,6.25]

b=[4.03,4.02,4.02,4.04,4.03,4.03]

data={"A":a,

"B":b}

df=pd.DataFrame(data) df.head()



##### Strength and Direction

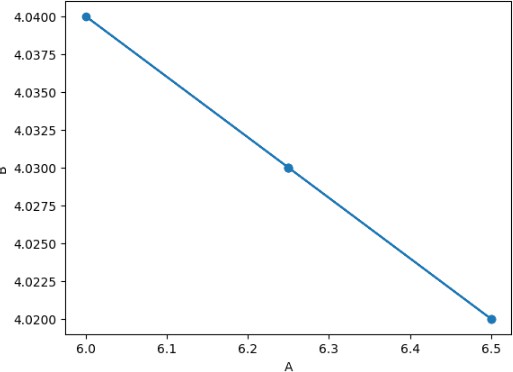
import statistics as stat corr=stat.correlation(df["A"],df["B"]) print(corr)

-1.0

##### Best Fitted Line

import matplotlib.pyplot as plt plt.scatter(df["A"],df["B"])

plt.plot(df["A"],df["B"]) plt.xlabel("A")

plt.ylabel("B") plt.show()

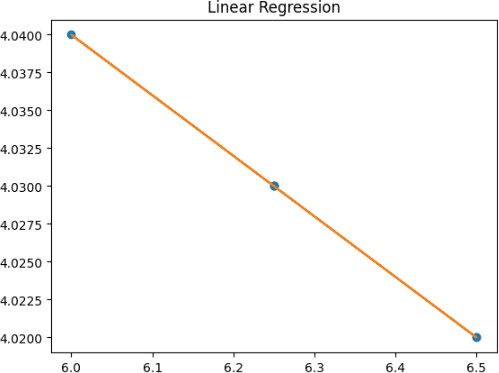
##### Linear Regression with Gradient Descent

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score from scipy import stats

slope, intercept, r, p, std\_err = stats.linregress(df["A"],df["B"]) def line(x):

return slope \* x + intercept model1 = list(map(line, df["A"])) plt.scatter(df["A"], df["B"])

plt.plot(df["A"],df["B"]) plt.title("Linear Regression") plt.plot(df["A"], model1) plt.show()



print(model1)

[4.03, 4.0200000000000005, 4.0200000000000005, 4.040000000000001, 4.03, 4.03]

mean\_squared\_error(model1,df["B"]) 3.944304526105059e-31

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score from scipy import stats

learning\_rate = 0.0001

iterations = 1000

slope = 0

intercept = 0 n = len(df)

for i in range(iterations):

y\_pred = slope \* df["A"] + intercept

d\_slope = (-2/n) \* sum(df["A"] \* (df["B"] - y\_pred)) d\_intercept = (-2/n) \* sum(df["B"] - y\_pred)

slope -= learning\_rate \* d\_slope intercept -= learning\_rate \* d\_intercept if i % 100 == 0:

mse = mean\_squared\_error(df["B"], y\_pred)

print(f"Iteration {i}: Slope = {slope:.4f}, Intercept = {intercept:.4f}, MSE = {mse:.4f}") def line(x):

return slope \* x + intercept

model = list(map(line, df["A"]))

slope, intercept, r, p, std\_err = stats.linregress(df["A"],df["B"]) def line(x):

return slope \* x + intercept model1 = list(map(line, df["A"]))

plt.figure(figsize=(10, 6)) plt.scatter(df["A"], df["B"])

plt.plot(df["A"],df["B"],color="green") plt.plot(df["A"], model1,color='blue') plt.scatter(df["A"], df["B"], label="Data Points")

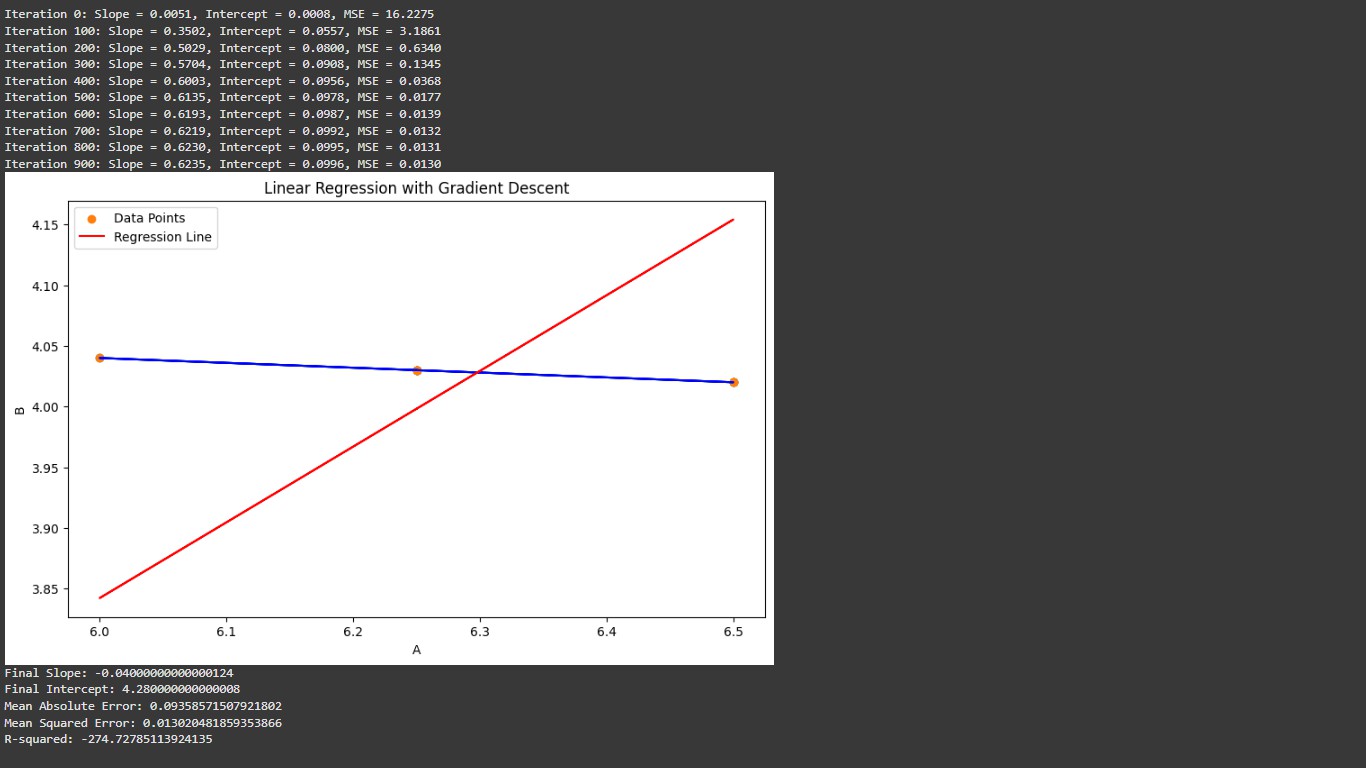
plt.plot(df["A"], model, color="red", label="Regression Line") plt.title("Linear Regression with Gradient Descent") plt.xlabel("A")

plt.ylabel("B") plt.legend() plt.show()

print("Final Slope:", slope) print("Final Intercept:", intercept)

mae = mean\_absolute\_error(df["B"], model) mse = mean\_squared\_error(df["B"], model) r2 = r2\_score(df["B"], model)

print("Mean Absolute Error:", mae) print("Mean Squared Error:", mse) print("R-squared:", r2)

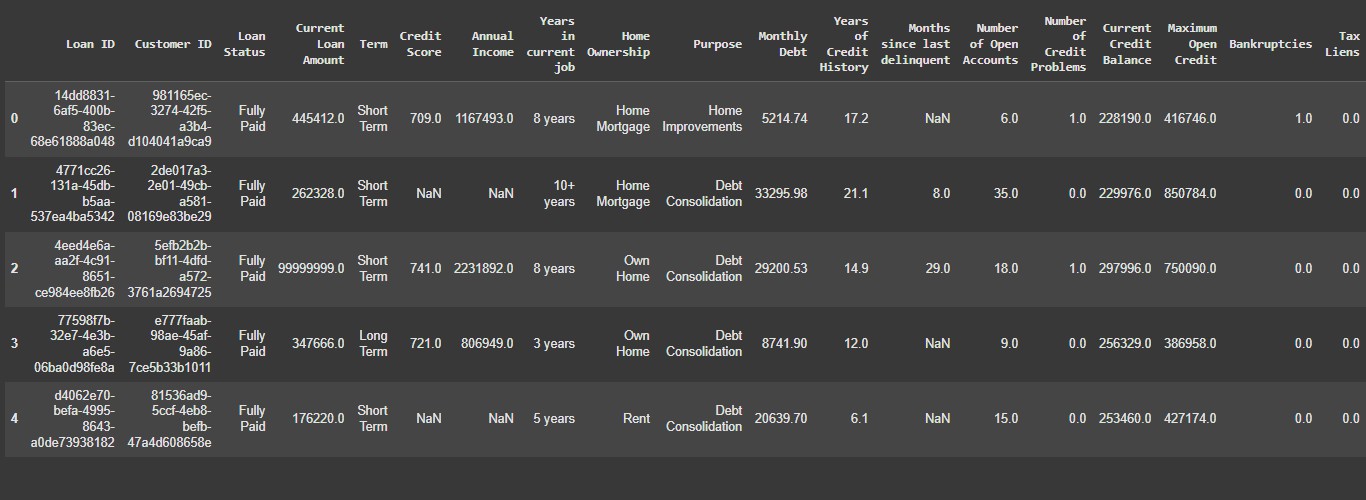


## Question 2:

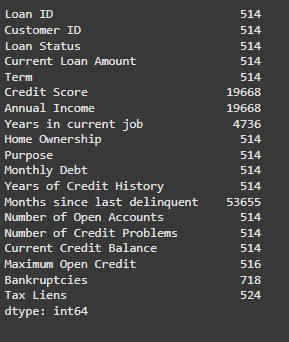
Consider the bank loan statement dataset - Show the bank loan score and make a category of good and bad customer

## Code:

loan=pd.read\_csv("./dataset/credit\_train.csv") loan.head()

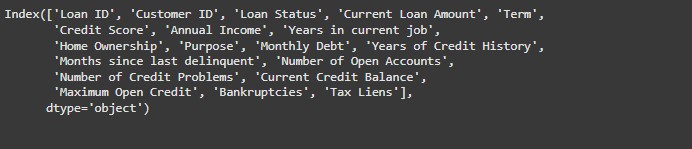


loan.isna().sum()

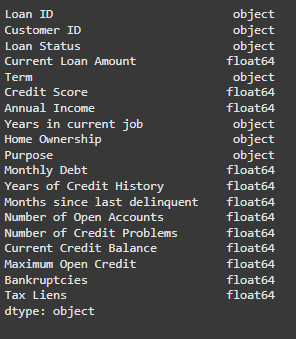


loan.dropna(inplace=True) loan.drop\_duplicates(inplace=True)

loan.isna().sum().sum()

loan.columns

test=pd.read\_csv("./dataset/credit\_test.csv") test.dropna(inplace=True) test.drop\_duplicates(inplace=True) loan.dtypes



for i in test.columns:

if i not in loan.columns: print(i)

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score from sklearn.preprocessing import LabelEncoder, StandardScaler

from sklearn.compose import ColumnTransformer from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer

Credit Score Prediction (Regression)

train=loan.drop(columns=["Loan ID", "Customer ID", "Loan Status"]) test\_data=test.drop(columns=["Loan ID", "Customer ID"])

X = train.drop(columns=["Credit Score"]) y = train["Credit Score"]

x\_test = test\_data.drop(columns=["Credit Score"]) y\_test = test\_data["Credit Score"]

numeric\_features = ['Current Loan Amount', 'Annual Income', 'Monthly Debt', 'Years of Credit History', 'Months since last delinquent',

'Number of Open Accounts', 'Number of Credit Problems', 'Current Credit Balance', 'Maximum Open Credit', 'Bankruptcies', 'Tax Liens']

categorical\_features = ['Term', 'Years in current job', 'Home Ownership', 'Purpose']

for col in categorical\_features: le = LabelEncoder()

X[col] = le.fit\_transform(X[col].astype(str)) x\_test[col] = le.fit\_transform(x\_test[col].astype(str))

numeric\_imputer = SimpleImputer(strategy='mean')

X[numeric\_features] = numeric\_imputer.fit\_transform(X[numeric\_features]) x\_test[numeric\_features] = numeric\_imputer.fit\_transform(x\_test[numeric\_features])

scaler = StandardScaler()

X[numeric\_features] = scaler.fit\_transform(X[numeric\_features]) x\_test[numeric\_features] = scaler.fit\_transform(x\_test[numeric\_features])

model2 = RandomForestRegressor(n\_estimators=100, random\_state=42)

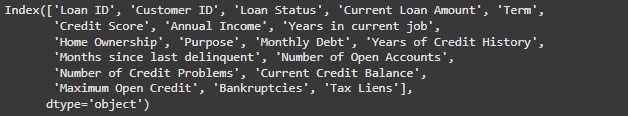
model2.fit(X,y)

y\_pred = model2.predict(x\_test) print(y\_pred) print(y\_test.values)

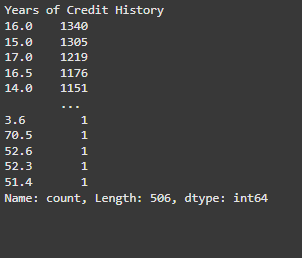


loandf=pd.read\_csv("./dataset/credit\_train.csv")

loandf.columns



loandf['Years of Credit History'].value\_counts()



Loan Status Prediction (Classification) import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split from sklearn.ensemble import RandomForestClassifier from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix from sklearn.preprocessing import LabelEncoder, StandardScaler

from sklearn.impute import SimpleImputer

df = pd.read\_csv("./dataset/credit\_train.csv") df.drop(columns=["Customer ID", "Loan ID"],inplace=True)

X = df.drop(columns=["Loan Status"]) y = df["Loan Status"].values

numeric\_features = ['Current Loan Amount', 'Credit Score', 'Annual Income', 'Monthly Debt', 'Years of Credit History',

'Months since last delinquent', 'Number of Open Accounts', 'Number of Credit Problems', 'Current Credit Balance', 'Maximum Open Credit', 'Bankruptcies', 'Tax Liens']

categorical\_features = ['Term', 'Years in current job', 'Home Ownership', 'Purpose'] numeric\_imputer = SimpleImputer(strategy='mean')

X[numeric\_features] = numeric\_imputer.fit\_transform(X[numeric\_features])

scaler = StandardScaler()

X[numeric\_features] = scaler.fit\_transform(X[numeric\_features])

for col in categorical\_features: le = LabelEncoder()

X[col] = le.fit\_transform(X[col].astype(str)) y = LabelEncoder().fit\_transform(y)

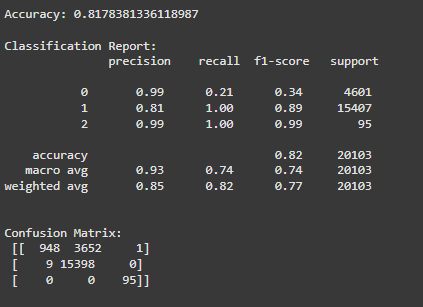
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

model = LogisticRegression(random\_state=42) model.fit(X\_train, y\_train)

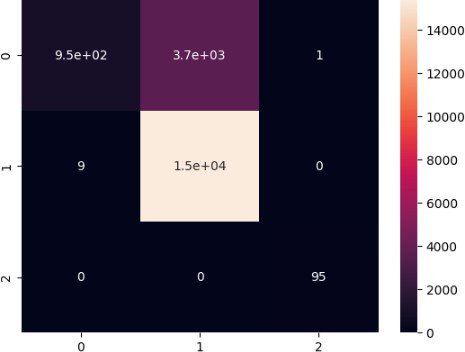
y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred) print("Accuracy:", accuracy)

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred)) print("\nConfusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))



import seaborn as sns sns.heatmap(confusion\_matrix(y\_test, y\_pred),annot=True)



df["Loan Status"].value\_counts()

