**UpGrad AIML Coding Questions and Solutions**

1. Implement a function to train a linear regression model using stochastic gradient descent (SGD) with mini-batch updates. The function should include options for different learning rates, batch sizes, and a regularization term.

import random

import numpy as np

def linear\_regression\_sgd(X, y, learning\_rate=0.01, batch\_size=1, num\_iterations=1000, regularization\_term=0.0):

n\_samples, n\_features = len(X), len(X[0])

# Initialize coefficients with zeros

coefficients = [0.0 for \_ in range(n\_features)]

for \_ in range(num\_iterations):

# Shuffle the data and create mini-batches

data = list(zip(X, y))

random.shuffle(data)

mini\_batches = [data[i:i+batch\_size] for i in range(0, n\_samples, batch\_size)]

for mini\_batch in mini\_batches:

X\_batch, y\_batch = zip(\*mini\_batch)

X\_batch, y\_batch = np.array(X\_batch), np.array(y\_batch)

# Compute the gradient of the loss function with respect to the coefficients

y\_pred = X\_batch.dot(coefficients)

error = y\_pred - y\_batch

gradient = X\_batch.T.dot(error) / batch\_size

# Add regularization to the gradient (excluding the intercept term)

gradient[1:] += (regularization\_term \* coefficients[1:]) / batch\_size

# Update the coefficients using the learning rate and gradient

coefficients -= learning\_rate \* gradient

return coefficients.tolist()

# Input data

X = [[1, 1], [1, 2], [1, 3], [1, 4]]

y = [3, 4, 5, 6]

learning\_rate = 0.01

batch\_size = 2

num\_iterations = 1000

regularization\_term = 0.1

# Train the linear regression model using stochastic gradient descent

coefficients = linear\_regression\_sgd(X, y, learning\_rate, batch\_size, num\_iterations, regularization\_term)

# Print the result

print("Optimized coefficients:", coefficients)

1. Write a function to implement linear regression with Lasso regularization (L1 regularization) using coordinate descent. The function should allow for different regularization parameters and tolerance levels for convergence.

import numpy as np

def soft\_thresholding(x, threshold):

if x > threshold:

return x - threshold

elif x < -threshold:

return x + threshold

else:

return 0.0

def coordinate\_descent\_lasso(X, y, regularization\_param, tolerance=1e-4, max\_iterations=1000):

n\_samples, n\_features = len(X), len(X[0])

coefficients = [0.0 for \_ in range(n\_features)]

prev\_coefficients = [0.0 for \_ in range(n\_features)]

for \_ in range(max\_iterations):

for j in range(n\_features):

# Update the j-th coefficient

X\_j = np.array([X[i][j] for i in range(n\_samples)])

y\_pred\_except\_j = np.dot(X, coefficients) - coefficients[j] \* X\_j

c\_j = np.dot(X\_j, y - y\_pred\_except\_j) / np.dot(X\_j, X\_j)

# Apply L1 regularization (Lasso) to update the coefficient

coefficients[j] = soft\_thresholding(c\_j, regularization\_param)

# Check for convergence

if np.linalg.norm(np.array(coefficients) - np.array(prev\_coefficients)) < tolerance:

break

prev\_coefficients = coefficients.copy()

return coefficients.tolist()

# Input data

X = [[1, 1], [1, 2], [1, 3], [1, 4]]

y = [3, 4, 5, 6]

regularization\_param = 0.1

tolerance = 0.001

# Train the linear regression model with Lasso regularization using coordinate descent

coefficients = coordinate\_descent\_lasso(X, y, regularization\_param, tolerance)

# Print the result

print("Optimized coefficients:", coefficients)

1. Create a program that performs logistic regression with L1 regularization (Lasso) using coordinate descent.

import numpy as np

def sigmoid(z):

return 1 / (1 + np.exp(-z))

def soft\_thresholding(x, threshold):

if x > threshold:

return x - threshold

elif x < -threshold:

return x + threshold

else:

return 0.0

def coordinate\_descent\_logistic(X, y, regularization\_param, tolerance=1e-4, max\_iterations=1000):

n\_samples, n\_features = len(X), len(X[0])

coefficients = [0.0 for \_ in range(n\_features)]

prev\_coefficients = [0.0 for \_ in range(n\_features)]

for \_ in range(max\_iterations):

for j in range(n\_features):

# Update the j-th coefficient

X\_j = np.array([X[i][j] for i in range(n\_samples)])

y\_pred\_except\_j = np.dot(X, coefficients) - coefficients[j] \* X\_j

c\_j = np.dot(X\_j, y - sigmoid(y\_pred\_except\_j)) / np.dot(X\_j, X\_j)

# Apply L1 regularization (Lasso) to update the coefficient

coefficients[j] = soft\_thresholding(c\_j, regularization\_param)

# Check for convergence

if np.linalg.norm(np.array(coefficients) - np.array(prev\_coefficients)) < tolerance:

break

prev\_coefficients = coefficients.copy()

return coefficients.tolist()

# Input data

X = [[1, 2], [2, 3], [3, 4], [4, 5]]

y = [0, 0, 1, 1]

regularization\_param = 0.1

tolerance = 0.001

# Train the logistic regression model with Lasso regularization using coordinate descent

coefficients = coordinate\_descent\_logistic(X, y, regularization\_param, tolerance)

# Print the result

print("Optimized coefficients:", coefficients)

1. Write a program to calculate the area under the ROC curve (AUC) for a logistic regression model.

import numpy as np

from sklearn.metrics import roc\_curve, auc

def calculate\_auc(y\_true, y\_pred):

fpr, tpr, \_ = roc\_curve(y\_true, y\_pred)

auc\_score = auc(fpr, tpr)

return auc\_score

# Given true class labels and predicted probabilities

y\_true = np.array([0, 1, 1, 0, 1])

y\_pred = np.array([0.2, 0.8, 0.6, 0.3, 0.9])

# Calculate AUC

auc\_score = calculate\_auc(y\_true, y\_pred)

print(f"AUC: {auc\_score:.3f}")

1. Write a program to calculate the log loss (binary cross-entropy) for a logistic regression model using vectorized operations.

import numpy as np

def log\_loss(y\_true, y\_pred):

epsilon = 1e-15 # To prevent log(0) cases

y\_pred = np.clip(y\_pred, epsilon, 1 - epsilon) # Clip predictions to avoid log(0) and log(1)

n = len(y\_true)

loss = -1/n \* np.sum(y\_true \* np.log(y\_pred) + (1 - y\_true) \* np.log(1 - y\_pred))

return loss

# Input data

y\_true = np.array([0, 1, 1, 0])

y\_pred = np.array([0.2, 0.8, 0.9, 0.3])

# Calculate log loss

log\_loss\_value = log\_loss(y\_true, y\_pred)

print("Log loss:", round(log\_loss\_value, 3))

1. Write a program to predict the class labels for new input data using a trained decision tree classifier.

import numpy as np

def predict\_sample(sample, tree):

if 'class' in tree:

return tree['class']

feature\_index = tree['feature\_index']

threshold = tree['threshold']

if sample[feature\_index] < threshold:

return predict\_sample(sample, tree['left'])

else:

return predict\_sample(sample, tree['right'])

def predict(X\_new, tree):

return [predict\_sample(sample, tree) for sample in X\_new]

# Example decision tree (output from the previous code)

decision\_tree = {'feature\_index': 0,

'threshold': 3,

'left': {'class': 0},

'right': {'class': 1}}

# Input data for prediction

X\_new = np.array([[1, 3], [1, 4], [2, 3]])

# Predict class labels using the trained decision tree classifier

predicted\_labels = predict(X\_new, decision\_tree)

print("Predicted class labels:", predicted\_labels)

1. Create a function to visualize a decision tree using a graph representation.

import numpy as np

import pydotplus

from sklearn.tree import export\_graphviz

from IPython.display import Image

def visualize\_decision\_tree(tree, feature\_names, class\_names):

dot\_data = export\_graphviz(tree, out\_file=None,

feature\_names=feature\_names,

class\_names=class\_names,

filled=True, rounded=True,

special\_characters=True)

graph = pydotplus.graph\_from\_dot\_data(dot\_data)

return graph

# Example decision tree (output from the previous code)

decision\_tree = {'feature\_index': 0,

'threshold': 3,

'left': {'class': 0},

'right': {'class': 1}}

# Input data

X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])

y = np.array([0, 0, 1, 1])

feature\_names = ['Feature 1', 'Feature 2']

class\_names = ['Class 0', 'Class 1']

# Visualize the decision tree

graph = visualize\_decision\_tree(decision\_tree, feature\_names, class\_names)

Image(graph.create\_png())

1. Write a program to perform hierarchical clustering using the complete linkage method.

import numpy as np

from scipy.cluster.hierarchy import linkage, fcluster

def hierarchical\_clustering(X, threshold, method='complete'):

# Calculate the distance matrix using the complete linkage method

distance\_matrix = linkage(X, method=method)

# Perform hierarchical clustering and get the cluster labels

cluster\_labels = fcluster(distance\_matrix, threshold, criterion='distance')

return cluster\_labels

# Input data

X = np.array([[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]])

# Threshold for forming clusters (you can adjust this value as needed)

threshold = 5

# Perform hierarchical clustering using the complete linkage method

cluster\_labels = hierarchical\_clustering(X, threshold, method='complete')

# Print the output

print("Cluster Labels:", cluster\_labels.tolist())

1. Implement a program to perform density-based clustering using the DBSCAN algorithm.

import numpy as np

from sklearn.cluster import DBSCAN

def density\_based\_clustering(X, epsilon, min\_samples):

# Initialize the DBSCAN clustering algorithm

dbscan = DBSCAN(eps=epsilon, min\_samples=min\_samples)

# Perform clustering and get the cluster labels

cluster\_labels = dbscan.fit\_predict(X)

return cluster\_labels

# Input data

X = np.array([[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]])

# Parameters for DBSCAN (epsilon and min\_samples)

epsilon = 3

min\_samples = 2

# Perform density-based clustering using the DBSCAN algorithm

cluster\_labels = density\_based\_clustering(X, epsilon, min\_samples)

# Print the output

print("Cluster Labels:", cluster\_labels.tolist())

1. Write a program to perform clustering using the fuzzy C-means algorithm.

import numpy as np

import skfuzzy as fuzz

def fuzzy\_c\_means\_clustering(X, c):

# Number of data points

n\_samples = X.shape[0]

# Number of features

n\_features = X.shape[1]

# Maximum number of iterations

max\_iter = 1000

# Fuzziness coefficient (m)

fuzziness = 2.0

# Termination threshold

epsilon = 1e-6

# Initialize the cluster centers randomly

cluster\_centers = np.random.rand(c, n\_features)

# Perform Fuzzy C-means clustering

cntr, u, \_, \_, \_, \_, \_ = fuzz.cluster.cmeans(

X.T, c, m=fuzziness, error=epsilon, maxiter=max\_iter, init=cluster\_centers

)

return cntr.T

# Input data

X = np.array([[1, 2], [2, 3], [10, 12], [11, 13], [20, 25], [22, 24]])

# Number of clusters (c) for Fuzzy C-means clustering

c = 3

# Perform Fuzzy C-means clustering

cluster\_centers = fuzzy\_c\_means\_clustering(X, c)

# Print the output

print("Cluster Centers:", cluster\_centers.tolist())