Implementation of the Research Paper:Open-environment Learning with Corruptions

Adity Banerjee(BS2304), Mrittika Giri(BS2332), Aman Verma(BS2309), Aryan Sahu(BS2317)

December 2023

1 Prerequisites I

The following are the necessary prerequisites for the first Algorithm in our research paper.

1. **SciPyA**: Python library which SciPy provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems.

2. **KD Tree**: It is a *space-partitioning data* structure for organizing points in a k-dimensional space,i.e., each data point has k properties.It is used for range searching in higher dimension(for example, in 1D range search we build a BST and split values in 1 dimension).The idea is that each level of the tree compares against 1 dimension.The **scipy.spatial.cKDTree** class provides an index into a set of k-dimensional points which can be used to rapidly look up the nearest neighbors(balls) of any point.

3. **Scikit-Learn**: also known as **sklearn** is a python library to implement machine learning models and statistical modellings.
   From **sklearn.decomposition** we import **PCA** which can be utilized for extracting information from a high-dimensional space by projecting it into a lower-dimensional sub-space. It tries to preserve the essential parts that have more variation of the data and remove the non-essential parts with fewer variation.PCA stands for Principal Component Analysis.

4. **Confidence intervals**: refers to the probability that a population parameter will fall between a set of values for a certain proportion of times.

5. **Query method**: The query() method allows you to query the DataFrame. The query() method takes a query expression as a string parameter, which has to evaluate to either True of False. It returns the DataFrame where the result is True according to the query expression.

2 Code for Algorithm 1

```
import numpy as np
from scipy.spatial import cKDTree
from scipy.stats import median_absolute_deviation
from sklearn.decomposition import PCA

def GRADIENTESTIMATOR(S, epsilon, dimension, confidence):
    # Implementation of GRADIENTESTIMATOR function

    # Your implementation here

    return estimated_gradient

def MEAN(S):
    # Implementation of MEAN function

    # Your implementation here

    return mean_value

def robust_ball(center, samples, epsilon):
    # Find the smallest ball containing (1 - epsilon) fraction of samples
    tree = cKDTree(samples)
    num_points = len(samples)
    num_inside = int(np.ceil((1 - epsilon) * num_points))
    distances, _ = tree.query(center, num_inside)
    radius = np.max(distances)
    return center, radius

def robust_mean_estimator(S, epsilon, dimension, confidence):
    d = dimension
    samples = np.array(S)

    if d == 1:
        return MEAN(samples)

    ball_centers = []
    for i in range(d):
        Si = samples[:, i]
        ci = np.median(Si)
        ball_centers.append(ci)

    ball_center, ball_radius = robust_ball(ball_centers, samples, epsilon)
    B = np.linalg.norm(samples - ball_center, axis=1) <= ball_radius
```

```
    Se = samples[B]

    if d == 1:
        return MEAN(Se)

    covariance_matrix = np.cov(Se, rowvar=False)
    pca = PCA(n_components=d)
    pca.fit(Se)
    V = pca.components_[:d // 2]
    W = pca.components_[d // 2:]

    SV = np.dot(Se, V.T)
    SW = np.dot(Se, W.T)

    mu_bV = GRADIENTESTIMATOR(SV, epsilon, d // 2, confidence)
    mu_bW = MEAN(SW)

    return mu_bV, mu_bW

# Example usage
gradient_sample_S = np.random.rand(100, 3)  # Replace with your actual data
level_of_corruption = 0.1
dimension_d = 3
confidence_delta = 0.95

result = robust_mean_estimator(gradient_sample_S,
level_of_corruption,dimension_d, confidence_delta)
print(result)
```

Please note that one needs to replace the placeholder data `gradient_sample_S` with his/her actual gradient sample data. Also, the implementation of the `GRADIENTESTIMATOR` and `MEAN` functions is missing and should be replaced with the specific implementation.

3 Prerequisites II

The following are the necessary prerequisites for the second Algorithm in our research paper.

1. **Numpy zeroes**: The NumPy zeros() function in Python is used to create an array of specified shapes and types, with all elements initialized to zero and similarly **np.ones()** creates an array with elements initialized to one.Data type of the returned array is float by default and in our program, np.ones() with dtype=int will return array with ones of integer

type.**np.ones(T,dtype=int)*10** will create an array with 'T' number of tens of integer type.

2. **Np.array**: np.array creates an array with any number of dimensions and when we use **np.sum()** with axis = 0, the function will sum over the 0th axis (the rows). It's basically summing up the values row-wise, and producing a new array (with lower dimensions).

3. **Numpy.random.randn**: The numpy.random.randn() function creates an array of specified shape and fills it with random values as per **standard normal distribution**.

We are describing a basic iterative optimization algorithm with a gradient estimator. Below is a Python code snippet that outlines the structure based on the description of the algorithm:

4 Code for Algorithm 2

```
import numpy as np

# Function to calculate gradient estimator
def GRADIENTESTIMATOR(St, epsilon, d, delta):
    # Implementation of your gradient estimator logic
    # ...

# Function for the optimization algorithm
def iterative_optimization(T, step_size_sequence,
sample_size_sequence,epsilon, dimension, confidence):
    # Initialization
    x = np.zeros(dimension)  # Assuming initial x1 is a zero vector

    for t in range(T):
        # Collect samples Zt from current position Pt
        Zt = np.random.randn(sample_size_sequence[t], dimension)
# Example random samples, replace with actual data collection

        St = np.sum(np.array([gradient_function(zi, x) for zi in Zt]), axis=0)
# Replace with your actual gradient calculation

        # Calculate gradient estimator
        get = GRADIENTESTIMATOR(St, epsilon, dimension, confidence)

        # Update x using the optimization step
        x=x - step_size_sequence[t]* get

    return x
```

```
# Example usage
T = 100  # Number of iterations
eta_sequence = np.linspace(0.1, 0.01, T)  # Example step size sequence
nt_sequence = np.ones(T, dtype=int) * 10  # Example sample size sequence
epsilon = 0.01  # Contaminated level
dimension = 2  # Dimension of the vector
confidence = 0.95  # Confidence level

result = iterative_optimization(T, eta_sequence, nt_sequence,
epsilon, dimension, confidence)
print("Final result:", result)
```

Please note that one needs to replace the placeholder data `gradient_sample_S` with his/her actual gradient sample data. Also, the implementation of the `GRADIENTESTIMATOR` and `MEAN` functions is missing and should be replaced with the specific implementation.

5 Acknowledgement

This is to extend our gratitude to Prof. Malay Bhattacharyya, who let us work on this project under his able guidance. This exposure has not only strengthened our basics in coding, but also helped us improvise our efficiency in comprehension and summarising of a research paper. We thoroughly benefited from the entire learning process, and look forward to such opportunities in the future.