Assignment – 2

Question3:

In the code, based on .py file given: I have taken all 4 different types of regressions to differentiate and analyze the epsilon-differential privacy across output perturbation:

Output Pertubation: Output perturbation is a post-processing differential privacy mechanism that protects sensitive data by injecting carefully calibrated noise into the outputs of a computation (e.g., model parameters, predictions, or statistics) before release. Unlike input or objective perturbation, it operates after model training. Given a deterministic function (e.g., a trained model) f:D->R over dataset D, output perturbation is explained as: M(D)=f(D)+Z

1. **LogisticRegressionWithL2Norm:**
   This implementation adds L2 regularization to standard logistic regression, using a penalty term lambda in the loss function. It includes numerical stability safeguards by clipping linear predictions (z = xθ) to [-500, 500] before exponentiation. The gradient computation incorporates both the logistic loss gradient and the L2 regularization term λθ. This version is particularly suited for privacy-preserving machine learning applications where regularization helps control sensitivity for differential privacy.
   Steps Involved:
   a. Adds L2 penalty
   b. Ensures numerical stability by clipping linear outputs
   c. Computes gradients combining: Standard logistic loss gradient and L2 regularization term
   d. Optimized for privacy by: Controlling parameter sensitivity
   Code Snippet:

```
class LogisticRegressionWithL2Norm:
  @staticmethod
  def loss(theta,x,y,lambda_param=None):
    n=x.shape[0]
    z=x.dot(theta.T)
    z=np.clip(z,-500,500)
    regularization=lambda_param/2*np.linalg.norm(theta,ord=2)**2
    objective_function=np.sum(np.log(1+np.exp(-y*z)))/n+regularization
    return objective_function
  @staticmethod
  def gradient(theta,x,y,lambda_param=None):
    n=x.shape[0]
    z=x.dot(theta.T)
    z=np.clip(z,-500,500)
    regularization=lambda_param*theta
    grad=-(np.transpose(x)@(y/(1+np.exp(y*z))))/n+regularization
    return grad
```

2. **LogisticRegression:**

This is a basic implementation of logistic regression without any regularization. It computes the average logistic loss across all samples and its corresponding gradient. The implementation is straightforward and efficient for standard classification tasks where regularization isn't required. It serves as a fundamental baseline for binary classification problems.

Steps Involved:

a. Basic binary classifier implementing the core logistic loss function
b. No regularization included (plain logistic regression)
c. Computing: Sigmoid-based probability estimates and Cross-entropy loss averaged across all samples
d. Gradient calculations

Code Snippet:

```
class LogisticRegression():
    @staticmethod
    def loss(theta, x, y, lambda_param=None):
        exponent = - y * (x.dot(theta))
        return np.sum(np.log(1 + np.exp(exponent))) / x.shape[0]
    @staticmethod
    def gradient(theta, x, y, lambda_param=None):
        exponent = y * (x.dot(theta))
        gradient_loss = - (np.transpose(x) @ (y / (1 + np.exp(exponent)))) / x.shape[0]
        gradient_loss = gradient_loss.reshape(theta.shape)
        return gradient_loss
```

3. **LogisticRegressionSinglePoint:**

Optimized for single-sample computations, this version calculates the loss and gradient for individual data points rather than batches. The loss function evaluates log(1 + exp(-yi xi theta)) per sample, and the gradient computes -(yi xi)/(1 + exp(yi xi theta)). This implementation is particularly useful for stochastic gradient descent (SGD) and differential privacy applications that require per-example gradient computations, such as DP-SGD.

Steps Involved:

a. Per-Instance Processing: Computes loss and gradients for individual data points
b. Loss calculation and Gradient Computation
c. Stochastic Gradient Descent (SGD) implementations

Code Snippet:

```
class LogisticRegressionSinglePoint():
    @staticmethod
    def loss(theta, xi, yi, lambda_param=None):
        exponent = - yi * (xi.dot(theta))
        return np.log(1 + np.exp(exponent))
    @staticmethod
    def gradient(theta, xi, yi, lambda_param=None):
        # Based on page 22 of
        # http://www.cs.rpi.edu/~magdon/courses/LFD-Slides/SlidesLect09.pdf
        exponent = yi * (xi.dot(theta))
```

```
return - (yi*xi) / (1+np.exp(exponent))
```

4. **LogisticRegression (Explicit L2):**

   This flexible implementation handles both single samples and batch inputs while supporting sparse matrix operations. It automatically reshapes inputs and gradients to work with sparse data structures like csr_matrix. Though it doesn't include explicit regularization in the shown code, its robust input handling makes it ideal for working with high-dimensional sparse datasets, such as those common in text classification and NLP tasks. The gradient computation is carefully designed to maintain compatibility with sparse matrix operations.

   Steps Involved:

   a. Enhanced Input Handling: Special optimization for sparse matrices
   b. Linear Transformation
   c. Loss Calculation and Gradient Computation

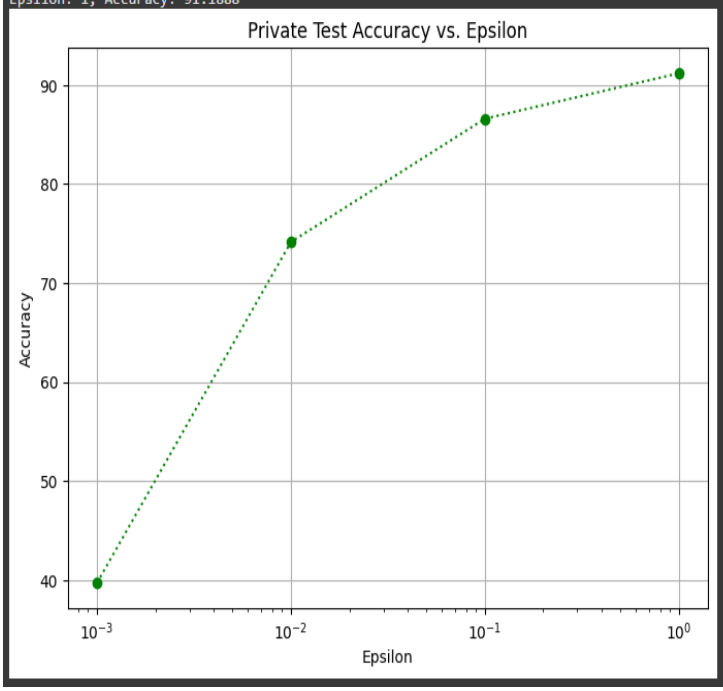
   Code Snippet:

```
class LogisticRegression():
    @staticmethod
    def loss(theta, x, y, lambda_param=None):
        #Loss function for logistic regression with without regularization
        exponent = - y * (x.dot(theta))
        return np.sum(np.log(1+np.exp(exponent))) / x.shape[0]
    @staticmethod
    def gradient(theta, x, y, lambda_param=None):
        x = x.reshape(1, -1) if x.ndim == 1 else x  # Reshape if it's 1D for a single sample
        # Ensure y is a scalar (for single sample) or a 1D array for multiple samples
        y = np.array(y).reshape(-1)  # Flatten y in case it's a scalar
        exponent = y * (x.dot(theta))
        #gradient_loss = - (np.transpose(x) @ (y / (1+np.exp(exponent)))) / (x.shape[0])
        #print(x.shape)
        #print(y.shape)
        #print(theta.shape)
        gradient_loss = - (x.T @ (y / (1 + np.exp(exponent)))) / x.shape[0]
        # Reshape to handle case where x is csr_matrix
        gradient_loss.reshape(theta.shape)
        return gradient_loss
```

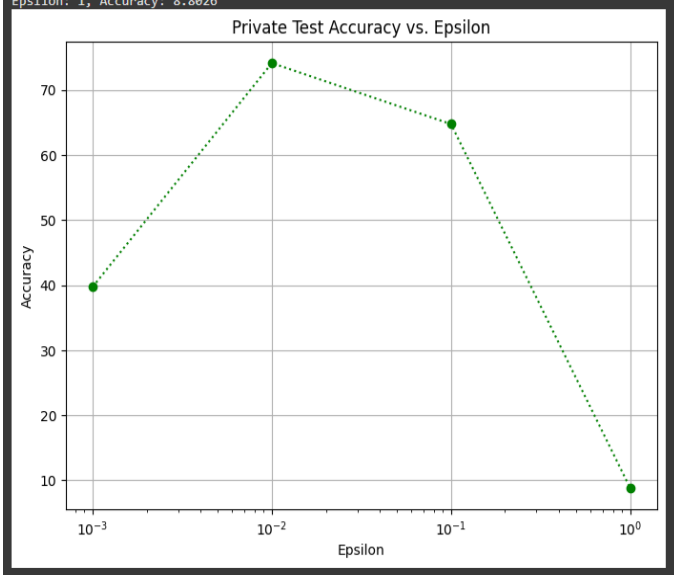
Analysis Charts:

As expected, lower ε (stronger privacy) typically reduces accuracy due to larger noise injection. Higher ε (weaker privacy) generally improves accuracy, approaching non-private baseline performance. Version1 of code the accuracy increases monotonically with epsilon. Version2 of code is peak at epsilon = 0.01 having accuracy 74.1% and then collapses at epsilon = 1 with accuracy 8.8%. Version3 of code is increasing except at epsilon = 0.01 with 96.4%. In version4 code the accuracy is fluctuating between 31.3% to 60.7%.

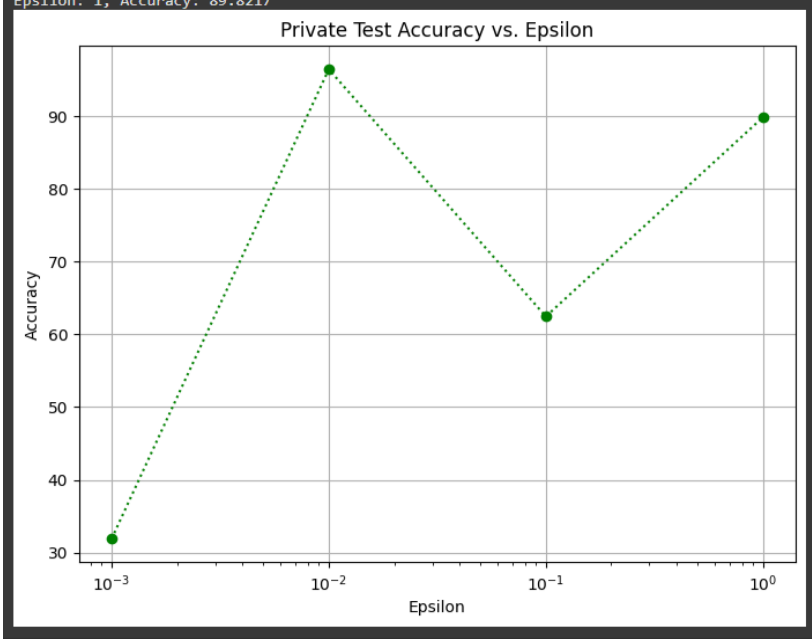
```
Epsilon: 0.001, Accuracy: 39.7489
Epsilon: 0.01, Accuracy: 74.1579
Epsilon: 0.1, Accuracy: 86.5990
Epsilon: 1, Accuracy: 91.1888
```



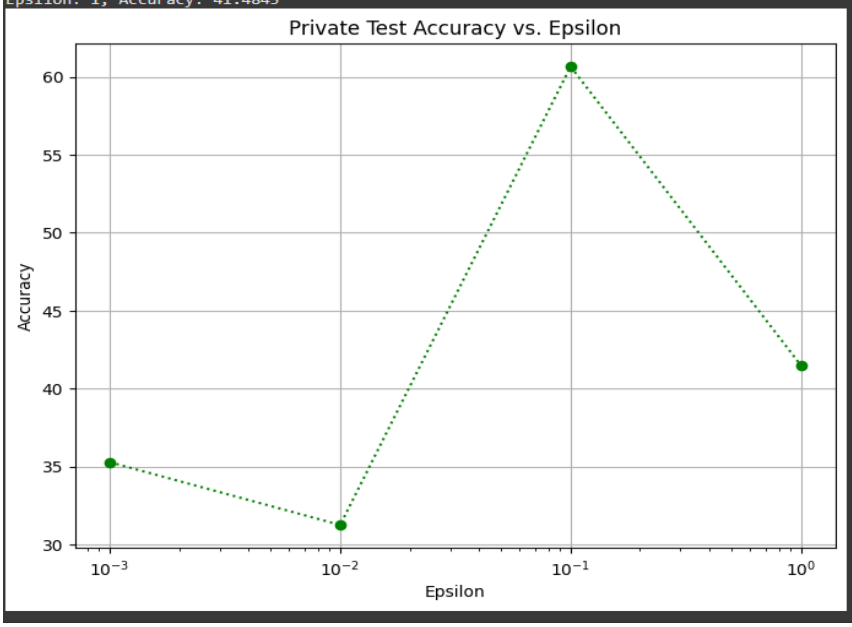

```
<ipython-input-2-621ed481c5f7>:12: RuntimeWarning: overflow encountered in exp
  gradient_loss = - (np.transpose(x) @ (y / (1 + np.exp(exponent)))) / x.shape[0]
Epsilon: 0.001, Accuracy: 39.7317
Epsilon: 0.01, Accuracy: 74.1150
Epsilon: 0.1, Accuracy: 64.7339
Epsilon: 1, Accuracy: 8.8026
```



```
  return - (y1*x1) / (1+np.exp(exponent))
Epsilon: 0.001, Accuracy: 31.9277
Epsilon: 0.01, Accuracy: 96.4130
Epsilon: 0.1, Accuracy: 62.4496
Epsilon: 1, Accuracy: 89.8217
```




```
Epsilon: 0.001, Accuracy: 35.2790
Epsilon: 0.01, Accuracy: 31.2505
Epsilon: 0.1, Accuracy: 60.6711
Epsilon: 1, Accuracy: 41.4845
```



Question4:

Objective Perturbation: Objective perturbation is a training-time differential privacy mechanism that preserves privacy by modifying the optimization objective of a machine learning model before training begins. Unlike output perturbation, it injects noise directly into the loss function rather than the final model parameters. Consider a Dataset D, Loss Function L(theta, D), privacy parameters – epsilon and delta, then Objective perturbation is L(theta, D) = L(theta, D) + Noise.

Steps Involved:

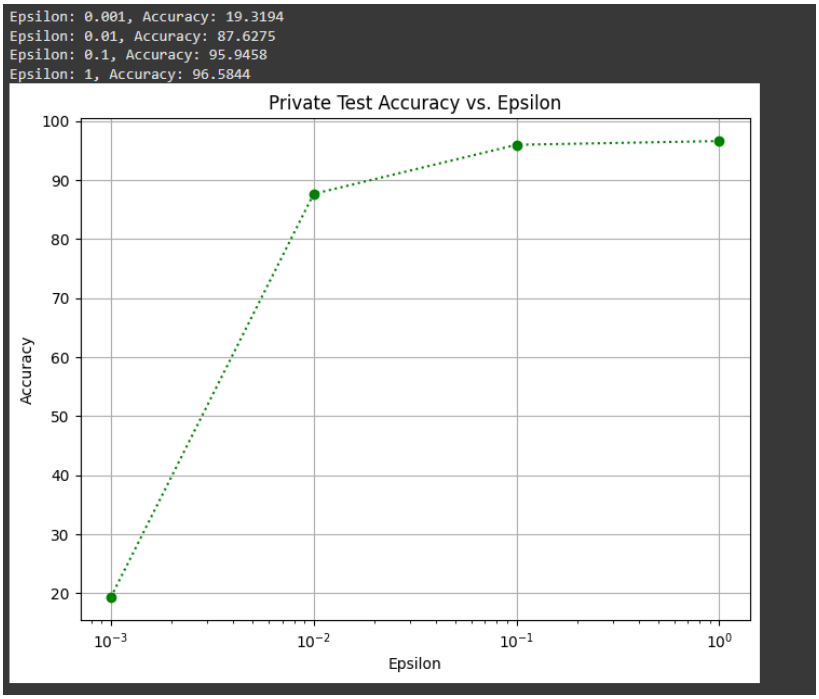
a. Noise Injection into the Objective Function: The loss() method involves a noise term is given as: objective_function = ... + np.dot(noise, theta). Here, noise is drawn from a Laplace distribution scaled to the privacy budget (epsilonP). This perturbs the linear term of the loss function, indirectly affecting the optimized weights.
b. Gradient Perturbation: The gradient() method adds the same noise to the gradients: grad = ... + noise.T. This ensures consistency between the perturbed objective and optimization.
c. Privacy Calibration: Noise scale (stdDev) is computed based on: Sensitivity (sensitivityVal=2): Bounds how much one data point can change the model. Next, Adjust the privacy budget (epsilonPPrime): Accounts for regularization effects. The code is given as: stdDev = sensitivityVal / epsilonPPrime; noiseVal = np.random.laplace(loc=0, scale=stdDev, size=numFeatures)
d. Strong Convexity Guarantee: The deltaVal term ensures the perturbed objective remains strongly convex: deltaVal = (paramC / (numSamples * (np.exp(epsilonPPrime/4) - 1))) – lambdaParam

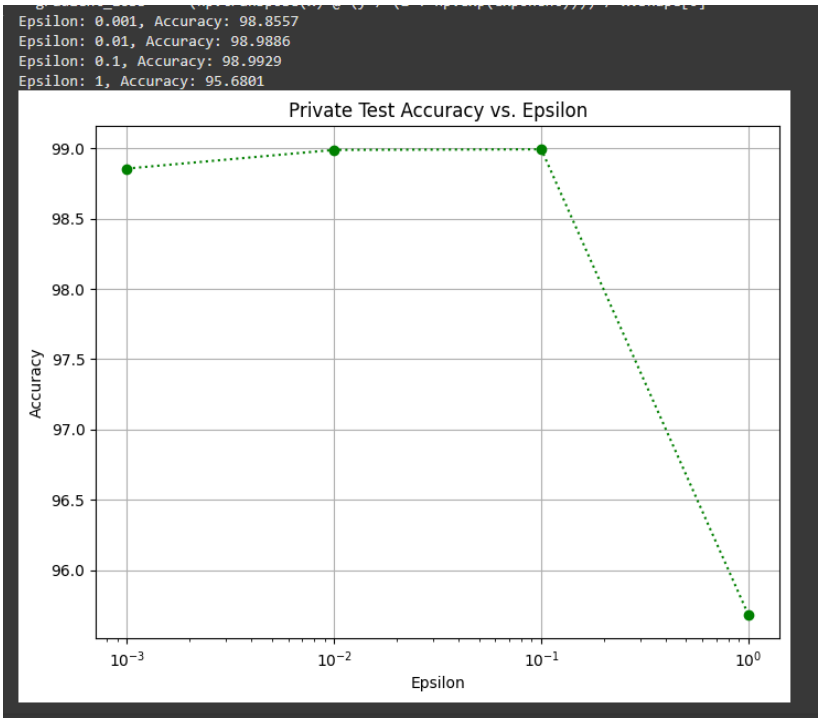
Workflow:

a. Data Preparation: Load and shuffle the KDDCup99 dataset.
b. Noise Generation: For each epsilonVal compute: epsilonPPrime, stdDev, noiseVal
c. Training: Optimize the perturbed objective using L-BFGS-B, The loss() and gradient() methods incorporate noise and L2 regularization.
d. Evaluation: Measure accuracy on the test set for each epsilon, Plot the privacy-utility tradeoff (accuracy vs. epsilon)

Analysis Charts:

The main chart – which is the first chart explains about objective perturbation. Strong privacy when epsilon = 0.001 yielding poor accuracy (19.3%) due to heavy noise injection. Moderate privacy when epsilon = 0.01 balances utility with accuracy (95.9%). Minimal privacy when epsilon = 1 approaches non private performance (96.6%). Here, Accuracy improves logarithmically with epsilon, showing diminishing returns.





Question5:

(1)

The Objective Perturbation Mechanism is designed to add noise to the objective function during the optimization process, aiming to provide differential privacy. However, in practice, this approach can fail to guarantee differential privacy for several reasons:

1. Dependence on Hyperparameters: The noise added to the objective function often depends on various hyperparameters (e.g., privacy budget epsilon, sensitivity of the objective function), and incorrect tuning of these hyperparameters could lead to insufficient noise. If the noise is too small, the model might inadvertently leak private information, thus violating differential privacy.
2. Inadequate Noise Calibration: Proper calibration of noise is crucial to ensuring privacy guarantees. If the added noise does not align with the true sensitivity of the objective function, it might not adequately mask the individual data points, which could lead to privacy violations.
3. Inability to Prevent Overfitting: While the objective perturbation mechanism adds noise, it may not effectively prevent overfitting, especially if the dataset is small or the model is complex. If overfitting occurs, the model may memorize private data, defeating the purpose of ensuring privacy.
4. Scaling Issues: In large-scale datasets, the mechanism may fail to add sufficient noise across the entire optimization process. As the dataset size increases, the scale of the perturbations may not be large enough to obscure individual data points, leading to potential leakage of private information.
5. Interaction with Optimizers: Many optimization algorithms (such as gradient-based methods) may amplify the effects of noise. In practice, the interaction between the added noise and the optimizer can lead to unexpected behaviors, making it difficult to control privacy leakage.

(2)

Although differential privacy typically imposes a trade-off with accuracy, in certain cases, differentially-private algorithms can outperform the non-private baseline. This can happen for several reasons:

1. Regularization Effect: Differentially private algorithms often act as a regularizer, reducing overfitting. By adding noise during training, these algorithms prevent the model from becoming too specific to the training data, which can improve generalization. This regularization effect can make the model more robust and lead to better performance on unseen data, especially in settings where the non-private model overfits.
2. Improved Generalization: The noise added to ensure privacy can help the model focus on broader patterns rather than memorizing specific training examples. This leads to better generalization when exposed to new, unseen data, thus improving accuracy in real-world applications. In some cases, this can make a differentially-private model outperform its non-private counterpart.
3. Noise as a Form of Data Augmentation: The added noise, while designed for privacy, can have the effect of augmenting the training data by introducing additional variability. This can help the model avoid memorizing specific data points and, as a result, can improve accuracy by creating a more diverse set of learned features.
4. Stabilization of Learning Process: Differentially private algorithms, particularly those that add noise at every step (e.g., differential privacy via gradient perturbation), can help stabilize the optimization process. This stability can sometimes prevent the model from overfitting, leading to more consistent and better-performing models.
5. Improved Handling of Noisy or Imperfect Data: In real-world scenarios, training data often contains noise and inconsistencies. Differentially private algorithms, through their regularizing effects, may be better at handling noisy or imperfect data, leading to better overall accuracy compared to non-private models that might overfit the noise.