Exploring Tradeoffs in Differential Privacy: An Empirical Study of Logistic Regression on Telemetry Data

Trey Scheid
tscheid@ucsd.edu

Tyler Kurpanek
tkurpane@ucsd.edu

Bradley Nathanson
bnathanson@ucsd.edu

Christopher Lum
cslum@ucsd.edu

Yu-Xiang Wang
yuxiangw@ucsd.edu

Abstract

Across many domains, telemetry data encodes personally identifiable information that could be used to discover sensitive information. In order to balance the tradeoff between the accuracy and privacy of models and queries, many DP algorithms exist. However, few studies help individuals decide which approach to take. We analyze 4 different DP methods for privatizing logistic regression, and their performance tradeoffs in practice on telemetry data. We found gradient descent to best minimize error for a set privacy budget, which achieved the lowest error of .120 for an epsilon of 2 across all DP methods.

Code: https://github.com/Brqdley/DSC180AProject

1 Introduction

1.1 Introduction

Differential privacy is a framework for data privacy that gives a mathematical guarantee that the sensitivity of each individual in the dataset is preserved. The core idea is to introduce random noise to the output of algorithms so that any single individual's data does not significantly affect the overall result. Mathematically, a mechanism is considered (,) differentially private if for all datasets D and D' which differ by at most 1 element when $\mathbb{P}[M(D) \in S] \leq e^{\epsilon}\mathbb{P}[M(D') \in S)] + \delta$ where ϵ and δ are privacy loss parameters. Smaller and imply stronger privacy guarantees.

Differential privacy is applied to algorithms, not datasets. One common and foundational algorithm is logistic regression. Many privatized implementations of logistic regression exist, leaving data scientists with a host of convoluted choices and complex language about parameters they may not fully understand. We hope to show some examples that will help practitioners implement this model on their own datasets.

1.2 Literature Review and Discussion of Prior Work

1.2.1 DP-SGD

This paper[1] investigates how privacy affects different mini-batch stochastic gradient descent algorithms for logistic regression classification. It is shown that privacy affects the batch size for optimal performance.

1.2.2 DP-ERM

This paper[2] focuses on general techniques to produce approximations of classifiers learned privately via empirical risk minimization (ERM). In doing so, it proposes a new privacy-preserving method, objective perturbation, which entails adding random noise to the function rather than adding noise directly to the data and does this before optimizing over classifiers. This method achieves better privacy-accuracy trade-offs and is applicable to regularized logistic regression which is what we are aiming to best privatize in our paper.

1.2.3 DP-Objective Perturbation

Similar to the last paper, this paper[3] highlights the objective perturbation mechanism with modern privacy analyses and computational tools. Specifically, it compares the mechanism

[1] https://cseweb.ucsd.edu/ kamalika/pubs/scs13.pdf
[2] https://jmlr.org/papers/volume12/chaudhuri11a/chaudhuri11a.pdf
[3] https://arxiv.org/abs/2401.00583

with differentially private stochastic gradient descent and outlines new methods of privacy accounting.

1.2.4 DP-FTRL

This paper Kairouz et al. (2021) designs and analyzes a DP variant of Follow-The-Regularized-Leader (DP-FTRL) that is comparable to or competes with DP-SGD. It is shown that finding a non-amplified model comparable to amplified DP-SGD is important for scenarios where precise requirements on shuffling and sampling are hard to obtain. DP-FTRL can outperform amplified DP-SGD at large values of , and be competitive for at all values with slight increases in computational cost.

1.3 Relevant Data

Differential privacy methods and guarantees are attractive for many domains. Telemetry is the remote data transfer of automated system measurements. As people use technology everyday their machines track usage diagnostics which are used by hardware and software manufacturers to reduce bugs and increase efficiency. System usage information is recorded at regular intervals and usually results in massive quantities of measurements. The identifiability of the specific machine or user of an event is a concern regardless of PIID tags. Dinur Nissim Dinur and Nissim (2003) and linkage attacks can be used to recover or reconstruct the original information: the source. This is a breach of privacy for a user which depending on the sensitivity of the information can be concerning. For example, personal laptops may send diagnostics to intel given that the user opts in to the program [Intel telemetry]. This program protects

The data that we will be making use of is a publicly available dataset on satellite telemetry [4]Bogdan (2024). The telemetry data was collected from the European Space Agency (ESA) OPS-SAT satellite from December 2019 to May 2024. This data has been useful for researchers in non-private, supervised anomaly detection (Ruszczak, et al. 2023, Ruszczak, et al. 2024). The data available consists of a small sample of the total data collected by the satellite selected by the operation engineers for anomaly detection (Ruszczak, et al. 2024). The raw data was sampled irregularly and difficult for machine learning algorithms to learn so several features were extracted from the sampled signal values such as mean, kurtosis, number of peaks, and more, for a total of 18 extracted features (Ruszczak, et al. 2024). This dataset is important as a tool for researchers because telemetry data is often private, confidential, or unlabeled, making it difficult to apply machine learning techniques right away. With our focus being on using private machine learning techniques, having a clean dataset allows us to focus on the algorithms after preprocessing.

[4]https://zenodo.org/records/12588359

2 Methods

2.1 Plan

We will dedicate one section to each of the private algorithms discussed in our literature review. The methods for each will cover their foundations, and how they can be applied to our problem.

2.2 Gradient Descent

Gradient descent, is a method to find the minimum value of a function by iteratively updating parameters in the direction of steepest descent. Each iteration computes the gradient of the loss function with respect to the model parameters. When the gradient is large, the parameter update is more significant, whereas a smaller gradient results in a smaller update.

2.3 SGD

SGD, or stochastic gradient descent, is a method of gradient descent which samples mini-batches and updates the gradient based on these mini-batches instead of the entire dataset. This often converges faster than GD as updates are faster. Each iteration is faster but the path towards the minimum is inherently more noisy, as the mini-batches can be substantially different from each other.

2.4 Output Perturbation

Output perturbation is an approach in differential privacy that involves adding calibrated noise to the final model parameters (theta) after training. In logistic regression specifically, the method works by first training a standard logistic regression model to completion, then adding Gaussian noise to the resulting model coefficients. The noise scale is inversely proportional to the regularization parameter λ, which helps control the sensitivity of the model. The L2 regularization term in your loss function is crucial as it ensures the optimization problem is strongly convex, which is a requirement for the privacy guarantees of output perturbation.

2.5 Objective Perturbation

Similar to output perturbation, objective perturbation is a fairly straightforward method of adding noise into logistic regression (or any generalized linear model). Objective perturbation adds noise into the loss function at the start of the optimization process, meaning that

out-of-the-box minimization techniques can be used to find weights that satisfy differential privacy. The modified loss function is then written as:

$$L^P(\theta; Z, b) = \sum_{z \in Z} l(\theta; z) + \frac{\lambda}{2}\|\theta\|_2^2 + b^T\theta \text{ where } b\ N(0, \sigma^2)$$

The privacy analysis used for objective perturbation was proposed by Redburg et al (2023) which we use for privacy accounting.

2.6 FTRL

2.6.1 Background

Follow the Leader is an optimization method which tackles online linear optimization problems. The algorithm stands out as a candidate for privatization due to the lack of sample amplification. This enables the algorithm to be implemented on federated learning problems without sacrifice to the utility or privacy.

Although other versions exist, for our applications the regularized implementation is ideal. Regularization was added to address the oscillation problem that was encountered in practice. A differentially private version of the algorithm is proposed by Guha Thakurta and Smith (2013), which uses approximation. Afterwards Kairouz et al. (2021) improved the theoretical bounds leading to our current implementation. Although many versions are possible, for the convex optimization problem (negative log loss) in logistic regression we will use the DP-FTL algorithm with regularization, momentum, mini-batches, and tree-restarts. Stronger utility can be achieved using newer noise structure: BLT McMahan, Xu and Zhang (2024).

2.6.2 Algorithm

Algorithm□ : DP-FTRLM: Differentially Private Follow-The-Regularized-Leader with Momentum

Require: Data $D = [d_1, \ldots, d_n]$, privacy parameters (ε, δ), learning rate η, momentum γ, L2 clip norm L, regularization λ

Initialize $\theta_0 = 0$, $v_0 = 0$

Initialize binary tree $\mathscr{T}$ with height $h = \lceil \log_2(n) \rceil + 1$

for $t = 1$ *to* n **do**

Get mini-batch gradients $\nabla_i = \nabla_\theta \ell(\theta_t; d_i)$

for *batch* **do**

Clip gradients: $\nabla_i^c = \nabla_i \cdot \min(1, \frac{L}{\|\nabla_i\|_2})$

Average clipped gradients: $g_t = \frac{1}{|B|} \sum i \in B \nabla_i^c$

Add g_t to tree at leaf node $2^{h-1} + t$

Get noisy prefix sum s_t by traversing up tree

Update momentum: $v_t = \gamma v_{t-1} + s_t$

Update model: $\theta_{t+1} = \arg\min_\theta \langle v_t, \theta \rangle + \frac{\lambda}{2} |\theta - \theta_0|^2$

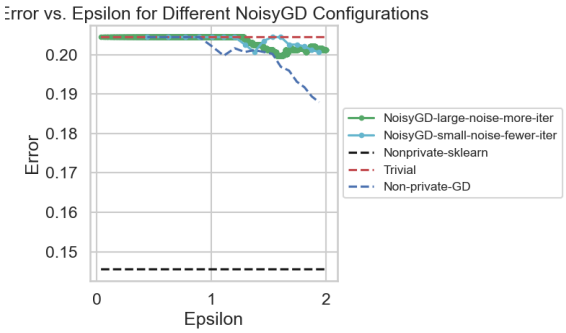
3 Results

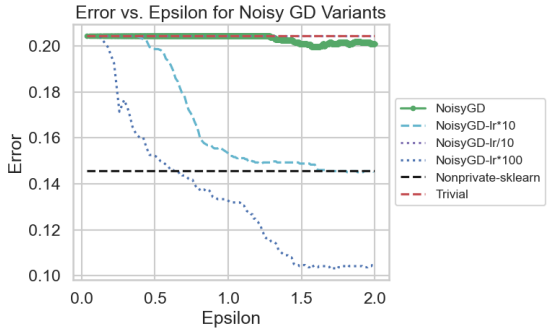
3.1 Baseline: Gradient Descent

We reference the AutoDP Logistic Regression notebook which displays plots that reflect Noisy GD with different sigma and learning rates. The results indicate that noise can lead to an optimal point with lower error.

From the plot below, we can see that the optimal configuration for GD is non private. But, for private GD both large noise with more iterations and small noise with fewer iterations converge very close. The GD with small noise has slightly less error, however, due to this and randomness we cannot conclude one is statistically significantly better than the other .

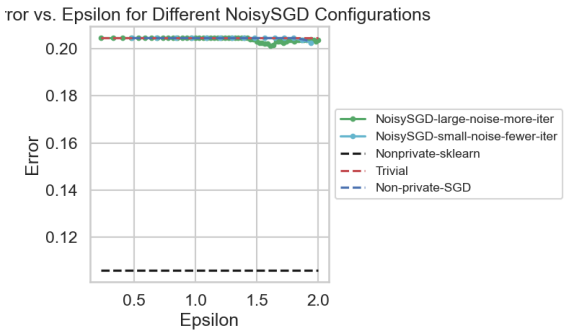
As we can see from the plot below, the optimal learning rate is the current learning rate configuration multiplied by 100. This learning rate converges below the non private logistic regression model which means the model performs very well. We can also see that with a learning rate multiplied by 10 it converges close to this non-private level. Compared to the normal learning rate, which converges close to the trivial level of .2, these perform much better. From this plot, we see these two different learning rates are a very good choice given a sigma of 30, learning rate of 1e-6, and batch size of 64.



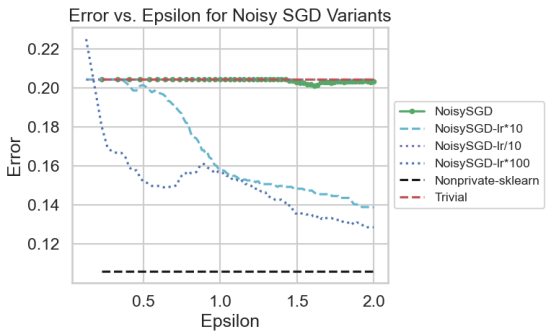


3.2 SGD

From the plot below, we can see that the optimal configuration for SGD is non private. Similar to GD, both large noise with more iterations and small noise with fewer iterations converge closely. However, it is a more significant difference and SGD with large noise and more iterations does seem to perform better overall. The learning does seem to be too small as the error at 2.00 Epsilon is less than .01 error different from the trivial error.



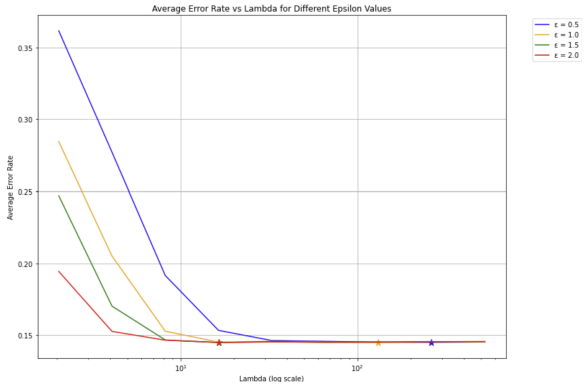
Once again the model converges at a lower rate with a learning rate of 10 or 100 times more than the default model for the selected parameters of a sigma of 30, learning rate of 1e-6, and batch size of 64. Compared to GD the error of the non private model is higher, which can be explained by the different baseline models used for SGD and GD, SGDClassifier and LogisticRegression respectively.



3.3 Output Perturbation

The code implements an experimental analysis of output perturbation in differentially private logistic regression, examining the relationship between privacy levels (ε), regularization strength (λ), and model performance. The experiment tests four privacy levels (ε = 0.5, 1.0, 1.5, 2.0) against varying regularization strengths, with λ values ranging from 1/4 to 64 times a base value. Each configuration undergoes 100 independent runs to ensure statistical reliability, with the results visualized in a logarithmic-scale plot. The purpose of this is to find the optimal lambda for each epsilon value.

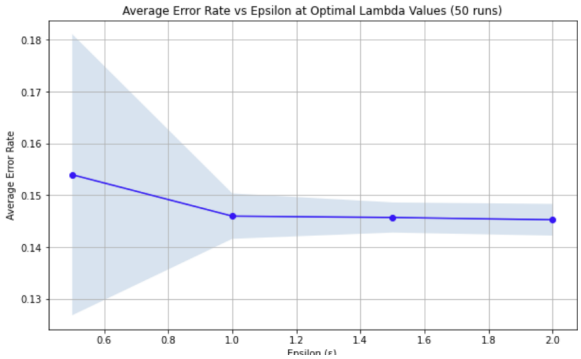
The results reveal a clear inverse relationship between privacy guarantees and optimal regularization strength. As the privacy constraint relaxes (ε increases), the optimal λ decreases. This stabilization suggests a threshold effect where additional privacy relaxation no longer significantly impacts the optimal regularization strength.



This code evaluates the performance of differentially private logistic regression across different privacy levels (ε) using the previously determined optimal lambda values. The experiment runs 50 iterations for each epsilon value (0.5, 1.0, 1.5, and 2.0) and visualizes the average error rates with a blue line. The shaded blue region represents one standard deviation above and below the mean, illustrating the variability in model performance across different runs.

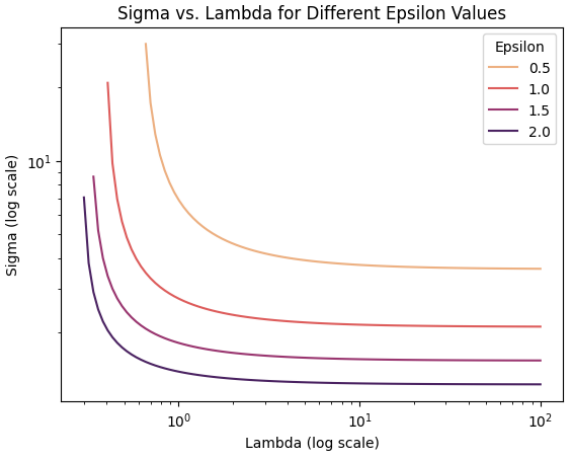
The results show a clear trend in error rates as privacy guarantees vary. With the strictest privacy setting (ε = 0.5), the model achieves an error rate of 0.1540, which improves to

0.1460 when ε increases to 1.0. Further increases in ε yield only marginal improvements, with error rates of 0.1457 and 0.1453 for ε values of 1.5 and 2.0 respectively.

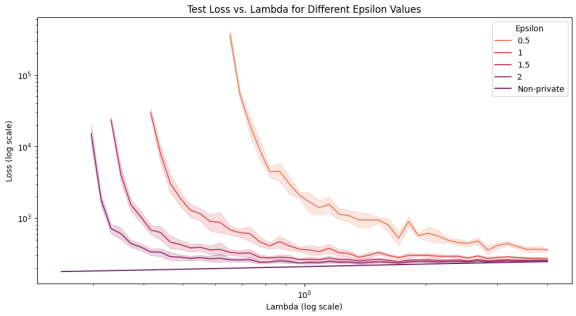


3.4 Objective Perturbation

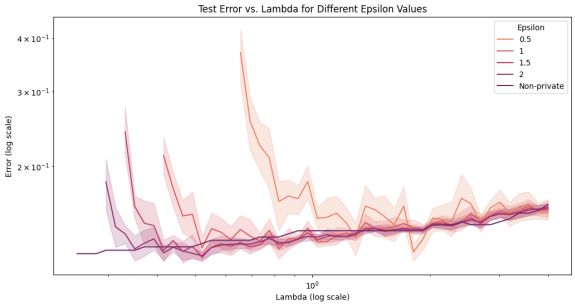
The below plot shows the relationship between sigma, lambda, and epsilon. Each of these values are tied together in the privacy analysis, as adding more variance will naturally make the mechanism less private, requiring more regularization to get the same privacy. As epsilon gets smaller, more noise must be added overall for each value of lambda. Another thing to note is that for small enough lambda for any given epsilon, there may be a point where the sigma required to achieve that epsilon reaches infinity and is not reachable. This plot will serve as the underlying justification for the following empirical plots.



The below plot shows the cross-entropy test loss on the non-private loss function (remember, objective perturbation minimizes a perturbed loss function) at the different privacy budgets (epsilon) .5, 1, 1.5, and 2 for different lambdas centered at 1 from beta to 4, as well as a baseline, non-private implementation. As we can see, smaller lambdas require larger and larger sigmas, meaning that more variance needs to be added, severely damaging the model's loss.

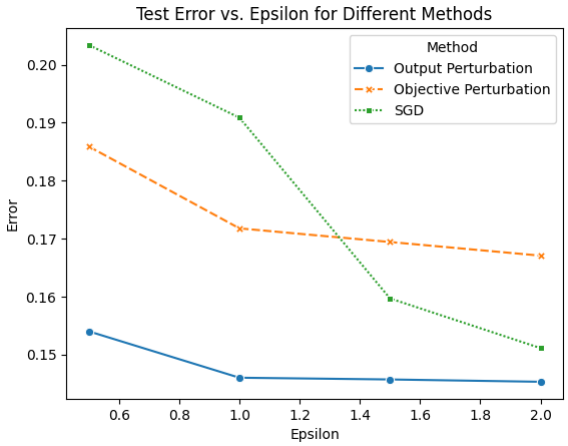


The plot below is similar to the plot above, except that it shows the test error instead of test loss. There is a negative logarithmic correlation between error and lambda. The non-private model has close to the lowest error for all lambda compared to the other epsilon values.



3.5 Combined Results

For our combined results, we can see that SGD has the lowest epsilon for all methods. Each method's error consistently decreases as epsilon increases.



4 Discussion

4.1 Interpretation

Our assumption for SGD and GD is that at low epsilon SGD performs better and at high epsilon GD works better. One is not inherently better than the other, as it depends on the specific privacy budget needed for the task.

However, when looking at the plots for SGD and GD, SGD outperformed GD for almost all learning rates, even converging faster to its minimum. This could be due to the specific parameters chosen, such as sigma.

Output Perturbation demonstrates strong performance across privacy budgets,showing a clear inverse relationship between privacy guarantees and optimal regularization strength. The results indicate that as privacy constraints are relaxed (higher ε values), the optimal regularization parameter λ decreases

With objective perturbation, privatized models compete fairly competitively with the non-private baseline with fairly low regularization. The additional noise in the objective function may serve as additional regularization which allows the model to generalize well to the test dataset.

4.2 Limitations

We only analyze a limited number of DP methods, which might not represent the full spectrum of available techniques. There could be better methods, or a combination of methods, that perform better than the ones we explored.

We also only use one telemetry dataset, which might not be representative of all types of sensitive data within telemetry, and not necessarily representative of the techniques' performance to different domains.

5 Conclusion

5.1 Summary

We implemented GD, SGD, Output Perturbation, Objective Perturbation, and FTRL, using AutoDP as the main python library. We created plots that show performance and relationships between different parameters (sigma, epsilon, lambda etc.).

For all of the techniques used, it is insightful to see the relationship between different parameters. It is also insightful to see the performance (error or loss) of different techniques at a range of epsilon.

The experimental results across different differential privacy methods reveal different pat-

terns and trade-offs. The findings show that Output Perturbation achieved the best performance with the lowest error rates, ranging from 0.1540 to 0.1453 across privacy budgets. Objective Perturbation demonstrates moderate performance with errors between 0.185882 and 0.167059, while SGD shows the highest initial error of 0.203391 but improves significantly to 0.151107 at $\varepsilon = 2.0$, indicating all methods perform better with relaxed privacy constraints.

5.2 Impact

These findings have direct applications for organizations handling sensitive telemetry data, particularly in satellite operations and hardware diagnostics systems. The research demonstrates that privacy-preserving techniques can be effectively implemented while maintaining essential analytical capabilities for system monitoring and anomaly detection.

5.3 Future Direction

We would like to further explore applying these techniques to other domains. As differential privacy is ever important with the amount of personal information being transacted and stored. There are many promising areas where these techniques can be used, such as healthcare and finance.

Ideally, we would like to add the mechanisms for each privatized logistic regression method to the differential privacy library we used, AutoDP, so future users can easily implement our mechanisms for their own use.

References

Bogdan, Ruszczak. 2024. "OPSSAT-AD - anomaly detection dataset for satellite telemetry." July. [Link]

Dinur, Irit, and Kobbi Nissim. 2003. "Revealing information while preserving privacy." In *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, NY, USA Association for Computing Machinery. [Link]

GuhaThakurta, Abhradeep, and Adam Smith. 2013. "(Nearly) Optimal Algorithms for Private Online Learning in Full-information and Bandit Settings." In *Advances in Neural Information Processing Systems*. Curran Associates, Inc. [Link]

Kairouz, Peter, Brendan McMahan, Shuang Song, OmThakkar, Abhradeep Thakurta, and Zheng Xu. 2021. "Practical and Private (Deep) Learning without Sampling or Shuffling." [Link]

McMahan, H.Brendan, Zheng Xu, and Yanxiang Zhang. 2024. "A Hassle-free Algorithm for Private Learning in Practice: Don't Use Tree Aggregation, Use BLTs." [Link]

DP_logistic_regression

December 9, 2024

[1]:
```
from sklearn.preprocessing import LabelEncoder
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
```

[2]:
```
df = pd.read_csv('data/dataset.csv')
```

[3]:
```
#convert categorical column to multiple binary
df=pd.get_dummies(df,columns=["channel"])
```

[4]:
```
from sklearn.linear_model import LogisticRegression
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import numpy as np

dataset = df
print('This is a regression dataset.')
print('Features are: ', "".join(dataset.columns))
print('The label is: ', "anomaly")
print('The shape of the data matrix iss', dataset.shape)

# Let's extract the relevant information from the sklearn dataset object
X = dataset[[x for x in dataset.columns if x!="anomaly"]]
y = dataset["anomaly"]

# -------------- Uncomment the following to test size = 0.9 when debugging you␣
 ↪code-------------
#
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.9,␣
 ↪random_state=93106)

# X = X_train
# y = y_train
#␣
 ↪-------------------------------------------------------------------------------␣
 ↪--------------
```

```
#can normalize

dim = X.shape[1]
n = X.shape[0]



# This is very important for DP linear regression.
# Also please make sure that the preprocessing does not use dataset for␣
 ↪training,
# e.g., the standard z-score cannot be used.



# Rescaling the feature vectors by their natural ranges (independent to the␣
 ↪data)
#re-scale correctly
#X = X @ np.diag(1./np.array([10,50,100,40,40000,1000,50,100]))
# This is to ensure that each feature is of the similar scale

# the following bounds are chosen independent to the data
x_bound = 1
y_bound = 1

# Preprocess the feature vector
X = x_bound*preprocessing.normalize(X, norm='l2')



def CE(score,y):
    # numerically efficient vectorized implementation of CE loss
    log_phat = np.zeros_like(score)
    log_one_minus_phat = np.zeros_like(score)
    mask = score > 0
    log_phat[mask] = - np.log( 1 + np.exp(-score[mask]))
    log_phat[~mask] = score[~mask] - np.log( 1 + np.exp(score[~mask]))
    log_one_minus_phat[mask] = -score[mask] - np.log( 1 + np.exp(-score[mask]))
    log_one_minus_phat[~mask] = - np.log( 1 + np.exp(score[~mask]))

    return -y*log_phat-(1-y)*log_one_minus_phat



def loss(theta):
    return np.sum(CE(X@theta,y))/n
```

```
def err(theta):
    return np.sum((X@theta > 0) != y) / n

def err_yhat(yhat):
    return np.sum((yhat != y)) / n


clf = LogisticRegression(random_state=0,fit_intercept=False).fit(X, y)
yhat = clf.predict(X)

err_nonprivate = err_yhat(yhat)
err_trivial = min(np.mean(y), 1-np.mean(y) )

# Nonprivate baseline
print('Nonprivate error rate is', err_yhat(yhat))

print('Trivial error rate is', err_trivial)
```

```
This is a regression dataset.
Features are:  segmentanomalytrainsamplingdurationlenmeanvarstdkurtosisskewn_pea
kssmooth10_n_peakssmooth20_n_peaksdiff_peaksdiff2_peaksdiff_vardiff2_vargaps_squ
aredlen_weightedvar_div_durationvar_div_lenchannel_CADC0872channel_CADC0873chann
el_CADC0874channel_CADC0884channel_CADC0886channel_CADC0888channel_CADC0890chann
el_CADC0892channel_CADC0894
The label is:  anomaly
The shape of the data matrix iss (2123, 31)
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel
Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)
instructions.
Nonprivate error rate is 0.14554875176636833
Trivial error rate is 0.20442769665567592
```

0.1 1. Let's first implement NoisyGD from scratch and represent it as a `Mechanism` in autodp.

We will start with the autodp representation of NoisyGD, which is a straightforward composition of Gaussian mechanisms. Then we will implement the algorithm itself.

[5]:
```
from autodp.autodp_core import Mechanism
from autodp.mechanism_zoo import GaussianMechanism
from autodp.transformer_zoo import ComposeGaussian


# The autodp Mechanism representation of NoisyGD is the following
class NoisyGD_mech(Mechanism):
    def __init__(self,sigma,coeff,name='NoisyGD'):
        Mechanism.__init__(self)
```

```
        self.name = name
        self.params={'sigma':sigma,'coeff':coeff}

        # ----------- Implement noisy-GD here with "GaussianMechanism" and␣
 ↪"ComposeGaussian" ----------------

        gm = GaussianMechanism(sigma,name='Release_gradient')
        # compose them with the transformation: ComposeGaussian.
        compose = ComposeGaussian()
        mech = compose([gm], [coeff])

        # ------------- return a Mechanism object named 'mech'␣
 ↪--------------------

        self.set_all_representation(mech)


# Now let's actually implement the noisy gradient descent algorithm

def gradient(theta):
    # -----------  Implement the gradient of f(theta) -----------
    grad = np.zeros(shape=(dim,))

    phat = np.exp(X@theta)/(1+np.exp(X@theta))
    grad = X[y==0,:].T@(phat[y==0]) -X[y==1,:].T@(1-phat[y==1].T)
    # ----------- Notice that f is the sum of the individual loss functions,␣
 ↪NOT the averge. -----------
    return grad




def GS_bound(theta):
    # ----------- Calculate the global sensitivity, given theta -------------
    # Note that you may start with a constant upper bound then consider using a␣
 ↪more adaptive bound

    GS = 100
    bound = np.linalg.norm(theta)
    GS = x_bound /(1+np.exp(-bound))

    GS_const = x_bound

    # ------------------------
    return GS
```

```
def run_NoisyGD_step(theta,sigma, lr):
    GS = GS_bound(theta)
    return theta - lr * (gradient(theta)  + GS*sigma*np.random.
 ↪normal(size=theta.shape))

# function to run NoisyGD
def run_NoisyGD(sigma,lr,niter, log_gap = 10):
    theta_GD = np.zeros(shape=(dim,))
    err_GD = []
    eps_GD = []
    for i in range(niter):
        theta_GD = run_NoisyGD_step(theta_GD,sigma, lr)
        if not i%log_gap:
            mech = NoisyGD_mech(sigma,i+1)
            eps_GD.append(mech.approxDP(delta))
            err_GD.append(err(theta_GD))
    return err_GD, eps_GD


# function to run NoisyGD
def run_nonprivate_GD(lr,niter, log_gap = 10):
    theta_GD = np.zeros(shape=(dim,))
    err_GD = []
    for i in range(niter):
        theta_GD = run_NoisyGD_step(theta_GD,0, lr)
        if not i%log_gap:
            err_GD.append(err(theta_GD))
    return err_GD
```

[6]:
```
theta = np.zeros(shape=(dim,))
ss =gradient(theta)
```

0.2 2. How do we choose the hyperparameters for NoisyGD?

Our strategy of choosing this hyperparameter is to first set the noise level and the number of iterations. (We can alternatively fix one of these and use autodp's privacy calibrator to determine the other.)

Once we decide on the noise level and the number of iterations, we will choose the learning rate by the optimal theoretical choice. It requires a data-dependent quantity (the gradient lipschitz constant is the largest eigenvalue of $X^T X$) which we wave our hand (ideally we can also privately release it at a small additional cost).

[7]:
```
from autodp.calibrator_zoo import eps_delta_calibrator

def find_appropriate_niter(sigma, eps,delta):
    # Use autodp calibrator for selecting 'niter'
    NoisyGD_fix_sigma = lambda x:  NoisyGD_mech(sigma,x)
```

```
    calibrate = eps_delta_calibrator()
    mech = calibrate(NoisyGD_fix_sigma, eps, delta, [0,500000])
    niter = int(np.floor(mech.params['coeff']))
    return niter


# Instantiate these parameters

def theoretical_lr_choice(beta_L,f0_minus_fniter_bound,dim,sigma,niter):
    # beta_L is the gradient lipschitz constant for the whole objective function
    # sigma is the variance of the gradient noise in each coordinate
    # niter is the intended number of iterations (the LR is optimized for the␣
 ↪point we get when finishing all niter)


    return np.minimum(1/beta_L,np.sqrt(2*f0_minus_fniter_bound / (dim *␣
 ↪sigma**2 *beta_L*niter)))


# You are supposed to find out what is the right choice of "beta_L" and␣
 ↪"f0_minus_fniter_bound"
```

0.3 Now let's run some experiments and plot the results!

We will first compare two regimes:
1. large noise, large number of iterations, small learning rate; 2. small noise, small number of iterations, large learning rate.

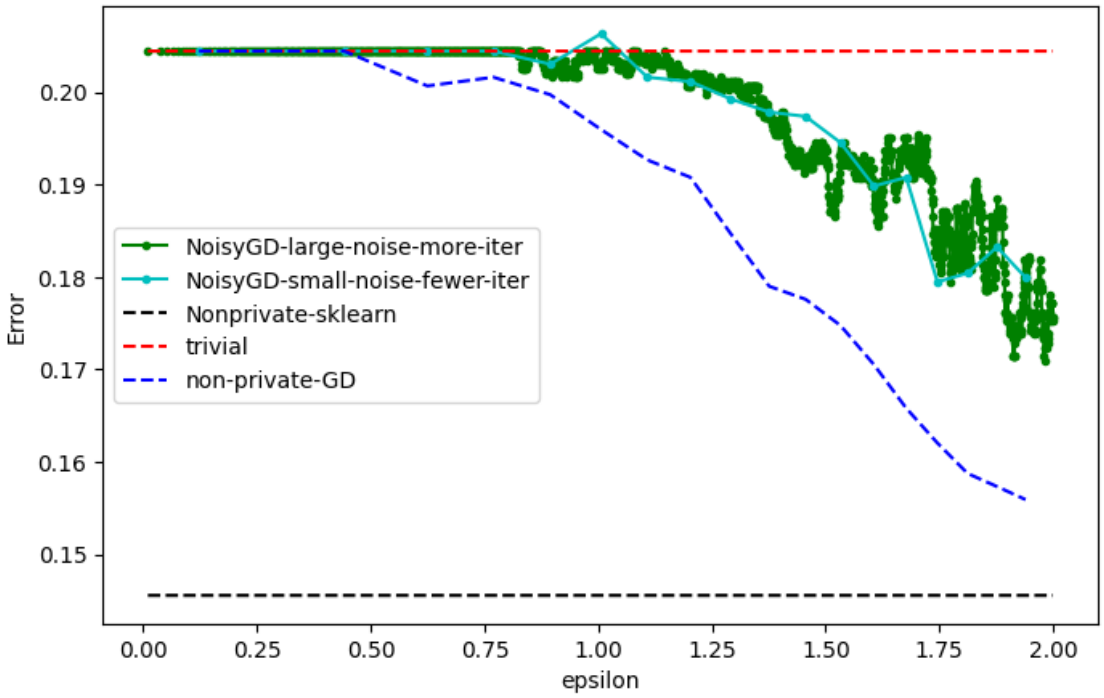
[8]:
```
# # find the theoretical learning rate choice by first working out the strong␣
 ↪smoothness property
# u,s,vT = np.linalg.svd(X.T@X)
# lambdamax = s[0]

# find the theoretical learning rate choice by first working out the strong␣
 ↪smoothness constant
beta = 1/4*n

f0_minus_fniter_bound  =  n*(-np.log(0.5))

GS = x_bound


# Large noise
sigma = 300.0
eps = 2.0
delta = 1e-6
niter = find_appropriate_niter(sigma, eps,delta)
```

```
print(niter)

lr = theoretical_lr_choice(beta,f0_minus_fniter_bound,dim,sigma*GS,niter)

err_GD1, eps_GD1 = run_NoisyGD(sigma,lr,niter)

# Small noise
sigma = 30
niter = find_appropriate_niter(sigma, eps,delta)
print(niter)
# niter = 500
lr = theoretical_lr_choice(beta,f0_minus_fniter_bound, dim,sigma*GS,niter)
err_GD2, eps_GD2 = run_NoisyGD(sigma,lr,niter)


# no noise baseline
err_GD0= run_nonprivate_GD(1/beta,niter)
```

```
18090
180
```

[ ]:

[9]:
```
## Let's also plot the results
import matplotlib.pyplot as plt
#%matplotlib inline
plt.figure(figsize=(8, 5))
#plt.plot(eps_GD2, err_GD0,'b.-')
plt.plot(eps_GD1, err_GD1,'g.-')
plt.plot(eps_GD2, err_GD2,'c.-')
plt.plot(eps_GD1,err_nonprivate*np.ones_like(eps_GD1),'k--')
plt.plot(eps_GD1,err_trivial*np.ones_like(eps_GD1),'r--')
plt.plot(eps_GD2,err_GD0,'b--')
#plt.ylim([0,0.1])

plt.
  ↪legend(['NoisyGD-large-noise-more-iter','NoisyGD-small-noise-fewer-iter','Nonprivate-sklear
plt.xlabel('epsilon')
plt.ylabel('Error')
plt.show()
```



So as we can see, the large-noise + more-iteration choice seems to converge faster on the epsilon scale, though they both arrive at very similar places towards the end of the training.

0.4 What if we wiggle the learning rate for a bit?

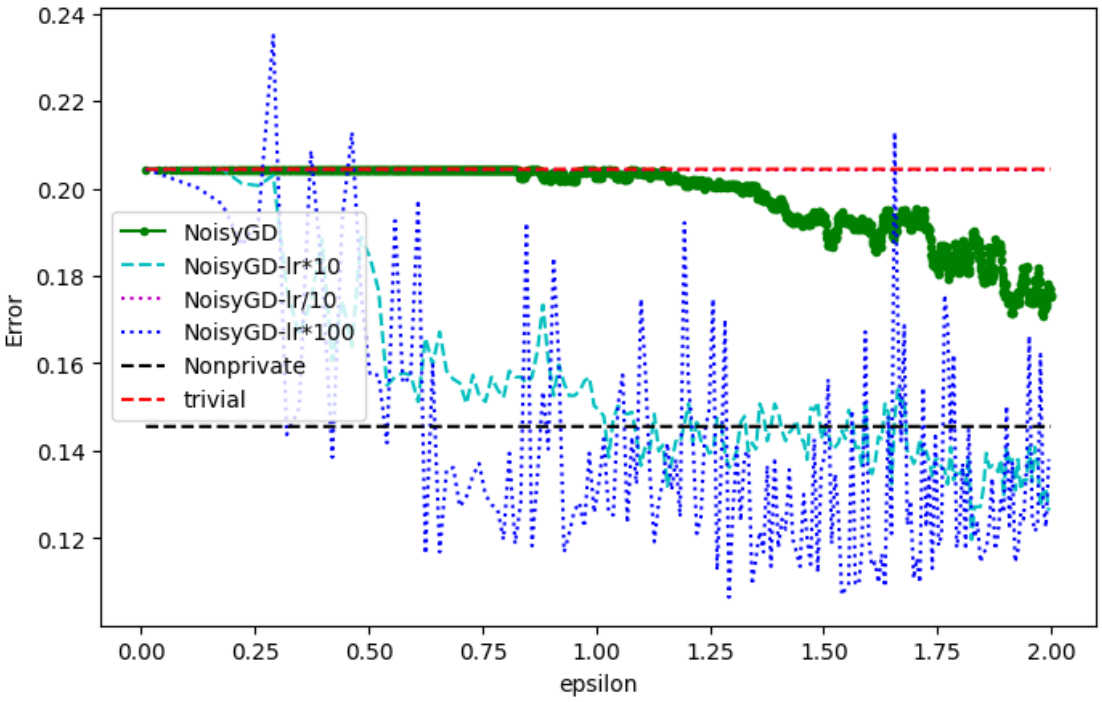
Next, we will consider the stability of the learning rate choices by trying larger and smaller learning rate near the theoretical choice:

3. Multiplying the learning rate by 10
4. Dividing the learning rate by 10

[10]:
```
sigma = 300.0
eps = 2.0
delta = 1e-6
niter = find_appropriate_niter(sigma, eps,delta)

lr = 10*theoretical_lr_choice(beta,f0_minus_fniter_bound,dim,sigma*GS,niter)
# Theoretical choice for GD  (those this is giving GD a bit of unfair advantage␣
 ↪because lambdamax is data-dependent)

err_GD3, eps_GD3 = run_NoisyGD(sigma,lr,niter,log_gap=100)

lr = 0.1*theoretical_lr_choice(beta,f0_minus_fniter_bound, dim,sigma*GS,niter)
# Theoretical choice for GD  (those this is giving GD a bit of unfair advantage␣
 ↪because lambdamax is data-dependent)
```

```
err_GD4, eps_GD4 = run_NoisyGD(sigma,lr,niter,log_gap=100)

lr = 100*theoretical_lr_choice(beta,f0_minus_fniter_bound,dim,sigma*GS,niter)
# Theoretical choice for GD  (those this is giving GD a bit of unfair advantage␣
 ↪because lambdamax is data-dependent)

err_GD5, eps_GD5 = run_NoisyGD(sigma,lr,niter,log_gap=100)
```

[11]:
```
# Let's also plot the median
import matplotlib.pyplot as plt
#%matplotlib inline
plt.figure(figsize=(8, 5))
# plt.plot(eps_budget_list,med_MSE_list[:,0],'o-')
# plt.plot(eps_budget_list,med_MSE_list[:,1],'s-')
plt.plot(eps_GD1, err_GD1,'g.-')
plt.plot(eps_GD3, err_GD3,'c--')
plt.plot(eps_GD4, err_GD4,'m:')
plt.plot(eps_GD5, err_GD5,'b:')
plt.plot(eps_GD1,err_nonprivate*np.ones_like(eps_GD1),'k--')
plt.plot(eps_GD1,err_trivial*np.ones_like(eps_GD1),'r--')


plt.legend(['NoisyGD','NoisyGD-lr*10','NoisyGD-lr/
 ↪10','NoisyGD-lr*100','Nonprivate','trivial'])
plt.xlabel('epsilon')
plt.ylabel('Error')
plt.show()
```


0.24
0.22
0.20
0.18
0.16
0.14
0.12
Error
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
epsilon
NoisyGD
NoisyGD-lr*10
NoisyGD-lr/10
NoisyGD-lr*100
Nonprivate
trivial

SGD

December 9, 2024

[1]:
```
import numpy as np
import pandas as pd
import warnings
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDClassifier
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import numpy as np
from autodp.autodp_core import Mechanism
from autodp.mechanism_zoo import GaussianMechanism
from autodp.transformer_zoo import ComposeGaussian
from autodp.autodp_core import Mechanism
from autodp.mechanism_zoo import ExactGaussianMechanism
from autodp.transformer_zoo import AmplificationBySampling, Composition
from autodp.calibrator_zoo import eps_delta_calibrator
```

[2]:
```
warnings.filterwarnings("ignore", category=RuntimeWarning)
np.random.seed(71)
df = pd.read_csv('data/dataset.csv')
```

[3]:
```
#convert categorical column to multiple binary
df=pd.get_dummies(df,columns=["channel"])
```

[4]:
```
dataset = df
print('This is a regression dataset.')
print('Features are: ', "".join(dataset.columns))
print('The label is: ', "anomaly")
print('The shape of the data matrix iss', dataset.shape)

# Let's extract the relevant information from the sklearn dataset object
X = dataset[[x for x in dataset.columns if x!="anomaly"]]
y = dataset["anomaly"]

# -------------- Uncomment the following to test size = 0.9 when debugging you␣
 ↪code-------------
#
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.9,␣
 ↪random_state=93106)
```

```
# X = X_train
# y = y_train
#␣
 ↪-------------------------------------------------------------------------------␣
 ↪--------------

#can normalize

dim = X.shape[1]
n = X.shape[0]



# This is very important for DP linear regression.
# Also please make sure that the preprocessing does not use dataset for␣
 ↪training,
# e.g., the standard z-score cannot be used.



# Rescaling the feature vectors by their natural ranges (independent to the␣
 ↪data)
#re-scale correctly
#X = X @ np.diag(1./np.array([10,50,100,40,40000,1000,50,100]))
# This is to ensure that each feature is of the similar scale

# the following bounds are chosen independent to the data
x_bound = 1
y_bound = 1

# Preprocess the feature vector
X = x_bound*preprocessing.normalize(X, norm='12')



def CE(score,y):
    # numerically efficient vectorized implementation of CE loss
    log_phat = np.zeros_like(score)
    log_one_minus_phat = np.zeros_like(score)
    mask = score > 0
    log_phat[mask] = - np.log( 1 + np.exp(-score[mask]))
    log_phat[~mask] = score[~mask] - np.log( 1 + np.exp(score[~mask]))
    log_one_minus_phat[mask] = -score[mask] - np.log( 1 + np.exp(-score[mask]))
    log_one_minus_phat[~mask] = - np.log( 1 + np.exp(score[~mask]))

    return -y*log_phat-(1-y)*log_one_minus_phat
```

```
def loss(theta):
    return np.sum(CE(X@theta,y))/n

def err(theta):
    return np.sum((X@theta > 0) != y) / n

def err_yhat(yhat):
    return np.sum((yhat != y)) / n


clf = SGDClassifier(random_state=0,fit_intercept=False).fit(X, y)
yhat = clf.predict(X)

err_nonprivate = err_yhat(yhat)
err_trivial = min(np.mean(y), 1-np.mean(y) )

# Nonprivate baseline
print('Nonprivate error rate is', err_yhat(yhat))

print('Trivial error rate is', err_trivial)
```

```
This is a regression dataset.
Features are:  segmentanomalytrainsamplingdurationlenmeanvarstdkurtosisskewn_pea
kssmooth10_n_peakssmooth20_n_peaksdiff_peaksdiff2_peaksdiff_vardiff2_vargaps_squ
aredlen_weightedvar_div_durationvar_div_lenchannel_CADC0872channel_CADC0873chann
el_CADC0874channel_CADC0884channel_CADC0886channel_CADC0888channel_CADC0890chann
el_CADC0892channel_CADC0894
The label is:  anomaly
The shape of the data matrix iss (2123, 31)
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel
Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX)
instructions.
Nonprivate error rate is 0.10598210080075365
Trivial error rate is 0.20442769665567592
```

0.1 1. Let's first implement NoisySGD from scratch and represent it as a `Mechanism` in autodp.

We will start with the autodp representation of NoisySGD, which is a straightforward composition of Gaussian mechanisms. Then we will implement the algorithm itself.

[5]:

```
class NoisySGD_Mechanism_With_Amplification(Mechanism):
    def __init__(self, prob, sigma, niter, PoissonSampling=True,␣
 ↪name='NoisySGD_with_amplification'):
        """
        A Noisy SGD mechanism with subsampling amplification.
        Args:
            prob: Sampling probability (batch_size / dataset_size).
            sigma: Noise scale for the Gaussian mechanism.
            niter: Number of SGD iterations.
            PoissonSampling: Whether Poisson sampling is used (default: True).
        """
        Mechanism.__init__(self)
        self.name = name
        self.params = {'prob': prob, 'sigma': sigma, 'niter': niter,
                       'PoissonSampling': PoissonSampling}

        subsample = AmplificationBySampling(PoissonSampling=PoissonSampling)

        gm = ExactGaussianMechanism(sigma=sigma)

        subsampled_mech = subsample(gm, prob, improved_bound_flag=True)

        compose = Composition()
        mech = compose([subsampled_mech], [niter])

        rdp_total = mech.RenyiDP

        self.propagate_updates(rdp_total, type_of_update='RDP')



# Now let's actually implement the noisy gradient descent algorithm

def stochastic_gradient(theta, X_batch, y_batch):
    phat = np.exp(X_batch @ theta) / (1 + np.exp(X_batch @ theta))
    grad = X_batch.T @ (phat - y_batch)
    #grad_norm = np.linalg.norm(grad)
    #if grad_norm > 1:
    #    grad = grad / grad_norm
    #
    return grad


def GS_bound(theta, batch_size):
    """
    Calculate global sensitivity for a mini-batch.
    """
```

```
    GS = 100
    bound = np.linalg.norm(theta)
    GS = x_bound / (1 + np.exp(-bound))
    return GS


def run_NoisySGD_step(theta, sigma, lr, X_batch, y_batch):
    """
    Perform one step of Noisy SGD.
    """
    GS = GS_bound(theta, batch_size=len(X_batch))
    noisy_gradient = stochastic_gradient(theta, X_batch, y_batch) + GS*sigma*np.
 ↪random.normal(size=theta.shape)
    return theta - lr * noisy_gradient

def run_NoisySGD(X, y, sigma, lr, niter, batch_size, log_gap=10):
    """
    Run stochastic gradient descent with noise for privacy.
    """
    theta_SGD = np.zeros(shape=(dim,))
    err_SGD = []
    eps_SGD = []
    prev=theta_SGD.copy()
    for i in range(niter):
        # Randomly sample a mini-batch
        idx = np.random.choice(len(y), batch_size, replace=False)
        X_batch = X[idx, :]
        y_batch = y[idx]

        # Perform a noisy SGD step
        theta_SGD = run_NoisySGD_step(theta_SGD, sigma, lr, X_batch, y_batch)
        prev=((prev*i)/(i+1))+(theta_SGD/(i+1))

        # Log error and privacy loss at intervals
        if i!=0:
            if not i % log_gap:
                mech = NoisySGD_Mechanism_With_Amplification(prob=batch_size /␣
 ↪len(y), sigma=sigma, niter=i + 1)
                eps_SGD.append(mech.approxDP(delta))
                #err_SGD.append(err(theta_SGD))
                err_SGD.append(err(prev))

    return err_SGD, eps_SGD

# function to run NoisyGD
def run_nonprivate_SGD(X, y, lr, niter, batch_size, log_gap):
    theta_SGD = np.zeros(shape=(dim,))
```

```
    prev=theta_SGD.copy()
    err_SGD = []
    eps_SGD = []
    for i in range(niter):
        # Randomly sample a mini-batch
        idx = np.random.choice(len(y), batch_size, replace=False)
        X_batch = X[idx, :]
        y_batch = y[idx]

        # Perform a noisy SGD step
        theta_SGD = run_NoisySGD_step(theta_SGD, 0, lr, X_batch, y_batch)
        prev=((prev*i)/(i+1))+(theta_SGD/(i+1))

        # Log error and privacy loss at intervals
        if i!=0:
            if not i % log_gap:
                err_SGD.append(err(prev))

    return err_SGD
```

[6]:
```
def find_appropriate_niter(sigma, eps,delta,num):
    # Use autodp calibrator for selecting 'niter'
    NoisyGD_fix_sigma = lambda x:  NoisySGD_Mechanism_With_Amplification(num/
 ↪2123, sigma, x)
    calibrate = eps_delta_calibrator()
    mech = calibrate(NoisyGD_fix_sigma, eps, delta, [1,500000])
    niter = int(np.floor(mech.params['niter']))
    return niter

# Instantiate these parameters

def theoretical_lr_choice(beta_L,f0_minus_fniter_bound,dim,sigma,niter):
    # beta_L is the gradient lipschitz constant for the whole objective function
    # sigma is the variance of the gradient noise in each coordinate
    # niter is the intended number of iterations (the LR is optimized for the␣
 ↪point we get when finishing all niter)


    return np.minimum(1/beta_L,np.sqrt(2*f0_minus_fniter_bound / (dim *␣
 ↪sigma**2 *beta_L*niter)))


# You are supposed to find out what is the right choice of "beta_L" and␣
 ↪"f0_minus_fniter_bound"
```

0.2 Now let's run some experiments and plot the results!

We will first compare two regimes:
1. large noise, large number of iterations, small learning rate; 2. small noise, small number of iterations, large learning rate.

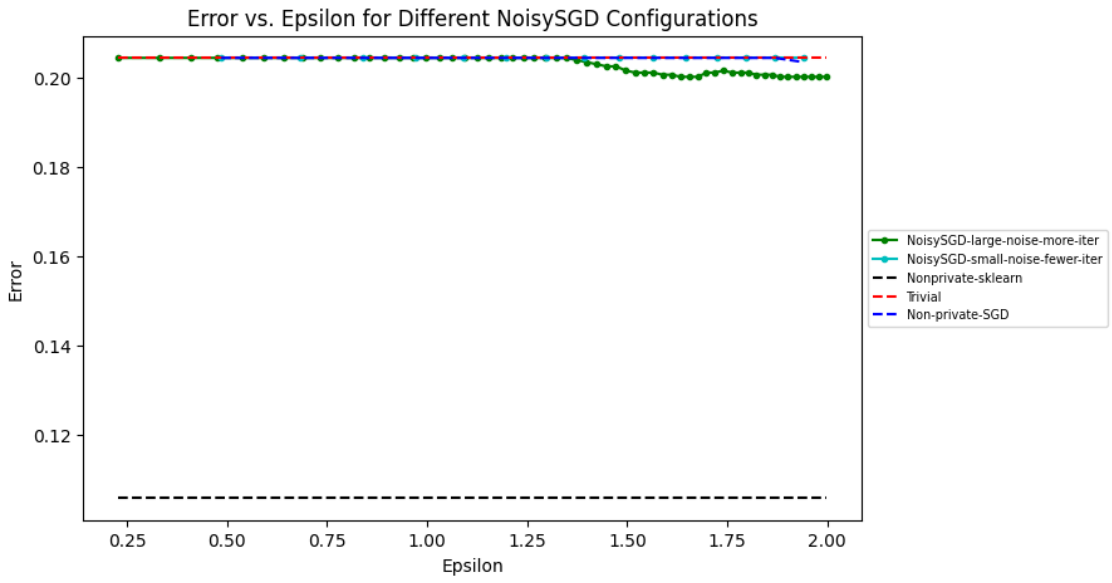
[7]:
```
# # find the theoretical learning rate choice by first working out the strong␣
 ↪smoothness property
# u,s,vT = np.linalg.svd(X.T@X)
# lambdamax = s[0]

# find the theoretical learning rate choice by first working out the strong␣
 ↪smoothness constant
beta = 1/4*n

f0_minus_fniter_bound  =  n*(-np.log(0.5))

GS = x_bound


# Large noise
sigma = 30.0
eps = 2
delta = 1e-6
niter=1000
niter = find_appropriate_niter(sigma, eps,delta,64)
lr = theoretical_lr_choice(beta,f0_minus_fniter_bound,dim,sigma*GS,niter)

err_SGD1, eps_SGD1 = run_NoisySGD(X,y,sigma,lr,niter,log_gap=3000,batch_size=64)
```

[8]:
```
# Small noise
sigma = 3
niter = 500
niter = find_appropriate_niter(sigma, eps,delta,64)
lr = theoretical_lr_choice(beta,f0_minus_fniter_bound, dim,sigma*GS,niter)
err_SGD2, eps_SGD2 = run_NoisySGD(X,y,sigma,lr,niter,log_gap=100,batch_size=64)
```

[9]:
```
#no noise baseline
err_SGD0=run_nonprivate_SGD(X, y, lr, niter, 64, log_gap=100)
```

[10]:
```
# Create the plot
plt.figure(figsize=(8, 5))
plt.plot(eps_SGD1, err_SGD1, 'g.-')
plt.plot(eps_SGD2, err_SGD2, 'c.-')
plt.plot(eps_SGD1, err_nonprivate * np.ones_like(eps_SGD1), 'k--')
plt.plot(eps_SGD1, err_trivial * np.ones_like(eps_SGD1), 'r--')
plt.plot(eps_SGD2, err_SGD0, 'b--')
```

```
# Add labels, legend, and optional limits
plt.legend([
    'NoisySGD-large-noise-more-iter',
    'NoisySGD-small-noise-fewer-iter',
    'Nonprivate-sklearn',
    'Trivial',
    'Non-private-SGD'
],fontsize='x-small',
    loc='center left',                # Place it to the left of the anchor point
    bbox_to_anchor=(1, 0.5),          # Anchor the legend to the right of the␣
 ↪plot
    ncol=1   )
plt.xlabel('Epsilon')
plt.ylabel('Error')
plt.title('Error vs. Epsilon for Different NoisySGD Configurations')

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



So as we can see, the large-noise + more-iteration choice seems to converge faster on the epsilon scale, though they both arrive at very similar places towards the end of the training.

0.3 What if we wiggle the learning rate for a bit?

Next, we will consider the stability of the learning rate choices by trying larger and smaller learning rate near the theoretical choice:

3. Multiplying the learning rate by 10
4. Dividing the learning rate by 10

[11]:
```
sigma = 30.0
eps = 2.0
delta = 1e-6
beta = 1/4*n
niter = 1000
GS = x_bound
f0_minus_fniter_bound  =  n*(-np.log(0.5))
niter = find_appropriate_niter(sigma, eps,delta,64)
```

[12]:
```
theoretical_lr=theoretical_lr_choice(beta,f0_minus_fniter_bound,dim,sigma*GS,niter)
```

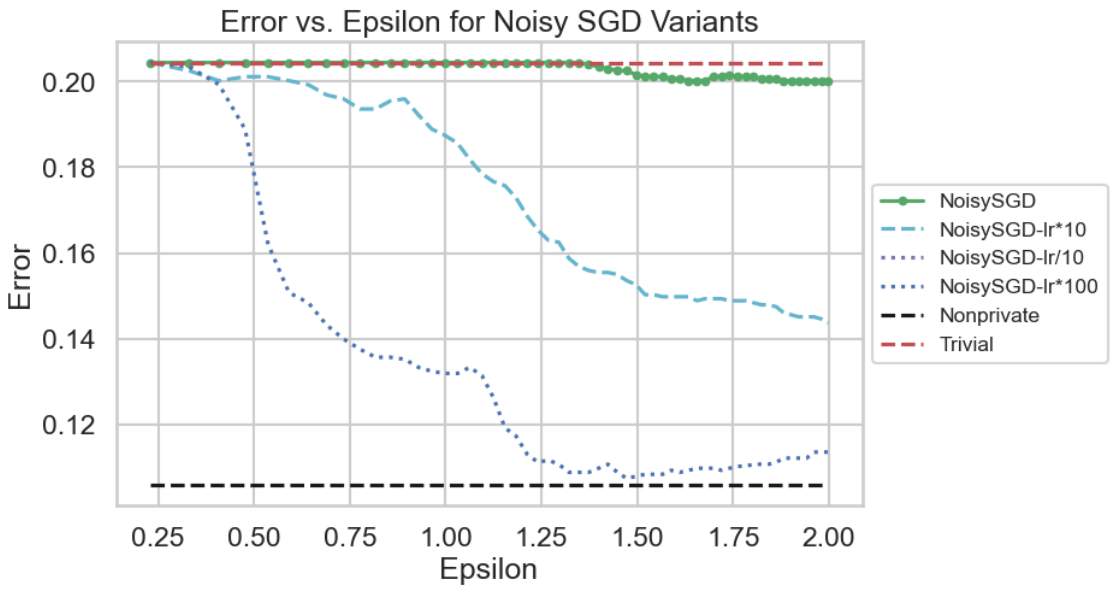
[13]:
```
lr = 10*theoretical_lr

err_SGD3, eps_SGD3 = run_NoisySGD(X,y,sigma,lr,niter,log_gap=3000,batch_size=64)
```

[14]:
```
lr = 0.1*theoretical_lr

err_SGD4, eps_SGD4 = run_NoisySGD(X,y,sigma,lr,niter,log_gap=3000,batch_size=64)
```

[15]:
```
lr = 100*theoretical_lr
err_SGD5, eps_SGD5 = run_NoisySGD(X,y,sigma,lr,niter,log_gap=3000,batch_size=64)
```

[16]:
```
# Set Seaborn style
import seaborn as sns
sns.set(style="whitegrid", context="talk")
sns.color_palette("flare")

# Create the plot
plt.figure(figsize=(8, 5))
plt.plot(eps_SGD1, err_SGD1, 'g.-')
plt.plot(eps_SGD3, err_SGD3, 'c--')
plt.plot(eps_SGD4, err_SGD4, 'm:')
plt.plot(eps_SGD5, err_SGD5, 'b:')
plt.plot(eps_SGD1, err_nonprivate * np.ones_like(eps_SGD1), 'k--')
plt.plot(eps_SGD1, err_trivial * np.ones_like(eps_SGD1), 'r--')

# Add legend, labels, and title
plt.legend(
    ['NoisySGD', 'NoisySGD-lr*10', 'NoisySGD-lr/10', 'NoisySGD-lr*100',␣
 ↪'Nonprivate', 'Trivial'],
    fontsize='x-small',
    loc='center left',                # Place it to the left of the anchor point
    bbox_to_anchor=(1, 0.5),          # Anchor the legend to the right of the␣
 ↪plot
    ncol=1                            # Single column layout
)
plt.xlabel('Epsilon')
```

```
plt.ylabel('Error')
plt.title('Error vs. Epsilon for Noisy SGD Variants')

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



[40]:
```
def run_NoisySGD_end(X, y, sigma, lr, niter, batch_size, log_gap=10):
    """
    Run stochastic gradient descent with noise for privacy and only one privacy␣
 ↪mech at the end
    """
    theta_SGD = np.zeros(shape=(dim,))
    err_SGD = []
    eps_SGD = []
    prev_theta=theta_SGD.copy()
    for i in range(niter):
        # Randomly sample a mini-batch
        idx = np.random.choice(len(y), batch_size, replace=False)
        X_batch = X[idx, :]
        y_batch = y[idx]

        # Perform a noisy SGD step
        theta_SGD = run_NoisySGD_step(theta_SGD, sigma, lr, X_batch, y_batch)
        #if i>niter*.1:
        #    theta_SGD=((prev_theta*i)/(i+1))+(theta_SGD/(i+1))
        # Log error and privacy loss at intervals
        if i==niter-1:
```

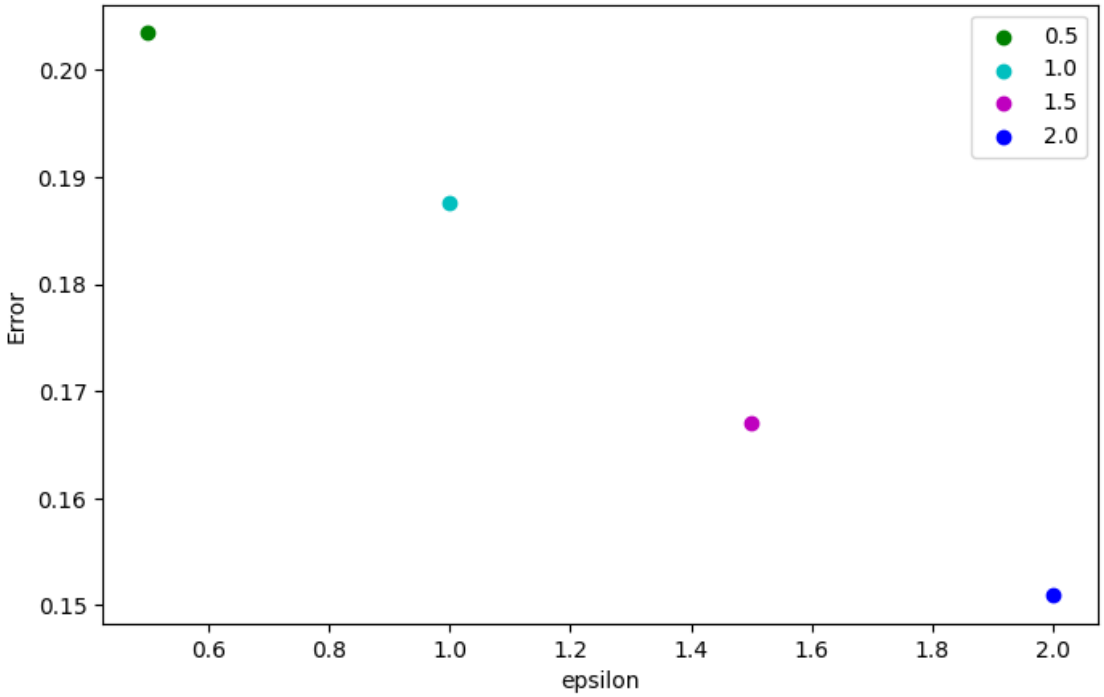
```
            mech = NoisySGD_Mechanism_With_Amplification(prob=batch_size /␣
 ↪len(y), sigma=sigma, niter=i + 1)
            eps_SGD.append(mech.approxDP(delta))
            err_SGD.append(err(theta_SGD))
    return err_SGD, eps_SGD
```

[17]:
```
ep=.05
#sigma epsilon delta batch
niter=find_appropriate_niter(30, ep,delta,64)
```

[42]:
```
#run averages
#batch size, sigma, and niter
                        #sigma, epsilon. delta, batch size
#pre-compute niter
def average_runs(num_runs, X, y, ep, delta, batch_size):
    """Run NoisySGD multiple times and return the averaged results."""
    err_list = []
    eps_list = []
    niter = find_appropriate_niter(3, ep, delta, batch_size)
    for _ in range(num_runs):
        err, eps = run_NoisySGD_end(X, y, 3, 0.01, niter, log_gap=1000,␣
 ↪batch_size=batch_size)
        err_list.append(err)
        eps_list.append(eps)

    # Compute the average across all runs
    avg_err = np.mean(err_list, axis=0)
    avg_eps = np.mean(eps_list, axis=0)
    return avg_err, avg_eps

# Number of runs to average
num_runs = 5

# Run and average for different epsilon (0.5, 1.0, 1.5, 2.0)
err_SGD_point5, eps_SGD_point5 = average_runs(num_runs, X, y, 0.5, delta, 64)
err_SGD_1point0, eps_SGD_1point0 = average_runs(num_runs, X, y, 1.0, delta, 64)
err_SGD_1point5, eps_SGD_1point5 = average_runs(num_runs, X, y, 1.5, delta, 64)
err_SGD_2point0, eps_SGD_2point0 = average_runs(num_runs, X, y, 2.0, delta, 64)
```

[43]:
```
plt.figure(figsize=(8, 5))

# Swap the variables: epsilon on x-axis, error on y-axis
plt.scatter(eps_SGD_point5, err_SGD_point5, color='g', label='0.5')
plt.scatter(eps_SGD_1point0, err_SGD_1point0, color='c', label='1.0')
plt.scatter(eps_SGD_1point5, err_SGD_1point5, color='m', label='1.5')
plt.scatter(eps_SGD_2point0, err_SGD_2point0, color='b', label='2.0')
```

```
plt.xlabel('epsilon')
plt.ylabel('Error')
plt.legend()

# Save the figure before showing it.
plt.show()
```



In [3]:
```
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from scipy.stats import norm
from scipy.optimize import root_scalar
import seaborn as sns
```

In [4]:
```
dataset = pd.read_csv('satellite.csv')
X = dataset.iloc[:, 5:].values
y = dataset.iloc[:, 1].values

dim = X.shape[1]
n = X.shape[0]

X = X @ np.diag(1./np.array([600, 500, 1e-1, 1e-2, 1e-2, 1, 1, 1, 6, 3, 90,

# clip data!
x_bound = 1
X = x_bound*preprocessing.normalize(X, norm='l2')

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar
```

In [5]:
```
def error(scores, labels):
  scores[scores > 0] = 1
  scores[scores <= 0] = 0
  return np.sum(np.abs(scores - labels)) / float(np.size(labels))

def non_private_loss(X, y, theta, lamb):
  log_likelihood = np.sum(y * (X @ theta) - np.log(1 + np.exp(X @ theta)))
  return -log_likelihood + lamb / 2 * np.linalg.norm(theta, 2) ** 2
```

In [6]:
```
def delta_(epsilon: float, beta: float, lambd: float, L: float, sigma: float
    eps_tilde = epsilon - np.abs(np.log(1 - beta / lambd))
    eps_hat = eps_tilde - L ** 2 / (2 * sigma ** 2)

    H = lambda sub: norm.cdf(-sub * sigma / L + L / (2 * sigma)) - np.exp(su

    return (
        2 * H(eps_tilde) if eps_hat >= 0
        else (1 - np.exp(eps_hat)) + np.exp(eps_hat) * 2 * H(L ** 2 / sigma
    )[0]

def epsilon_(target_delta: float, beta: float, lambd: float, L: float, sigma
    def objective(epsilon):
        return delta_(epsilon, beta, lambd, L, sigma) - target_delta

    # Use a scalar root solver
    try:
        result = root_scalar(objective, bracket=(1e-5, 100), method='brentq'
    except ValueError:
```

```
        return np.inf
    if result.converged:
        return result.root
    else:
        raise ValueError("Solver did not converge")

def sigma_(epsilon: float, target_delta: float, beta: float, lambd: float, L
    def objective(sigma):
        return delta_(epsilon, beta, lambd, L, sigma) - target_delta

    # Solve for sigma using a scalar root solver
    try:
        result = root_scalar(objective, bracket=(1e-1, 1e+3), method='brentc
    except ValueError:
        return np.inf
    if result.converged:
        return result.root
    else:
        raise ValueError("Solver did not converge for sigma")

def lambda_(epsilon: float, target_delta: float, beta: float, L: float, sigm
    def objective(lambd):
        return delta_(epsilon, beta, lambd, L, sigma) - target_delta

    # Solve for lambda using a scalar root solver
    try:
        result = root_scalar(objective, bracket=(beta + 1e-4, 100), method='
    except ValueError:
        return np.inf
    if result.converged:
        return result.root
    else:
        raise ValueError("Solver did not converge for lambda")
```

In [ ]:

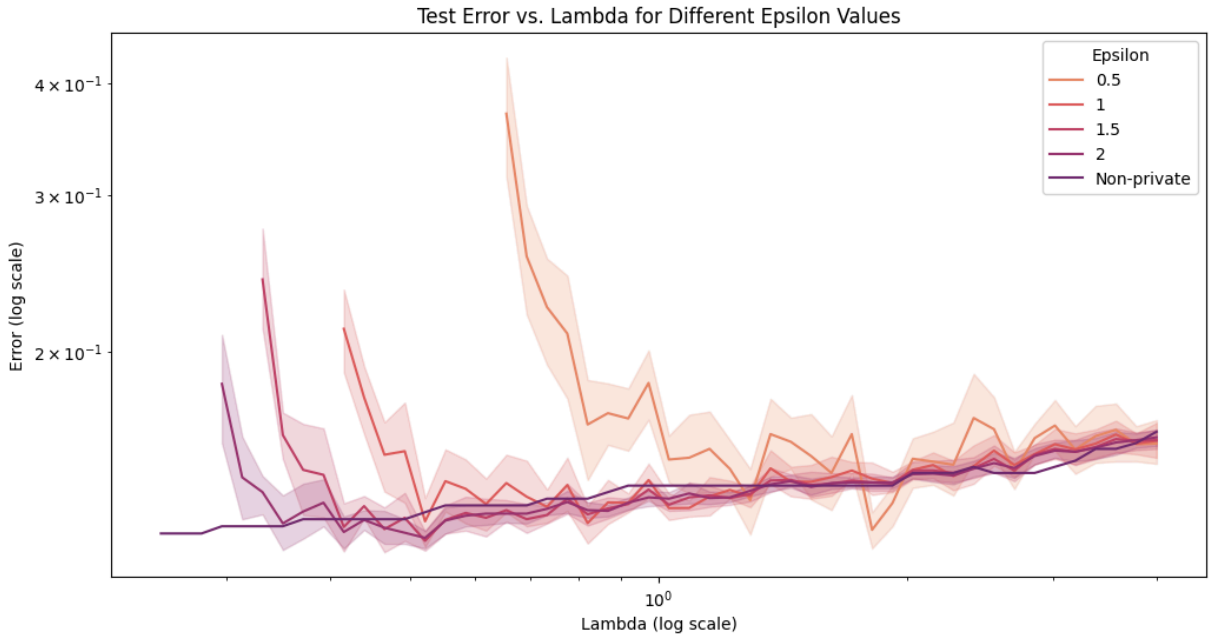
```
np.random.seed(42)

eps = [0.5, 1, 1.5, 2]
delt = 1e-6

L = x_bound
beta = x_bound ** 2 / 4
lambda_min = beta
lamb = cp.Parameter(nonneg=True)
sigma = cp.Parameter(nonneg=True)
theta = cp.Variable(dim)

log_likelihood = cp.sum(cp.multiply(y_train, X_train @ theta) - cp.logistic(

data = []

trials = 50
repeats = 10
lambda_vals = np.logspace(np.log10(lambda_min + 1e-4), np.log10(4), trials)
for i in range(trials):
    lamb.value = lambda_vals[i]
```

```
    for _ in range(repeats):
        unscaled_b = np.random.normal(0, 1, dim)

        for epsilon in eps:
            try:
                sigma.value = sigma_(epsilon, delt, beta, lamb.value, L)
            except ValueError:
                continue

            objpert_rand = (unscaled_b * sigma.value) @ theta
            problem = cp.Problem(
                cp.Maximize(log_likelihood
                            - lamb / 2 * cp.norm(theta, 2) ** 2
                            - objpert_rand
                ))

            try:
                problem.solve(solver=cp.ECOS)
            except cp.error.SolverError:
                print("SolverError at lambda = ", lamb.value, " and epsilon
                continue
            if theta.value is None:
                print("Didn't converge at lambda = ", lamb.value, " and epsi
                continue
            train_error = error((X_train @ theta).value, y_train)
            test_error = error((X_test @ theta).value, y_test)
            train_loss = non_private_loss(X_train, y_train, theta.value, lam
            test_loss = non_private_loss(X_test, y_test, theta.value, lamb.v

            data.append([epsilon, lamb.value, train_error, train_loss, 'trai
            data.append([epsilon, lamb.value, test_error, test_loss, 'test']

    # train_errors[epsilon] = train_error
    # test_errors[epsilon] = test_error
    # theta_vals[epsilon] = theta_vals_eps
    # lambda_vals[epsilon] = lambda_vals_eps
non_private_problem = cp.Problem(
    cp.Maximize(log_likelihood
                - lamb / 2 * cp.norm(theta, 2) ** 2
    ))

epsilon = "Non-private"

for i in range(trials):
    lamb.value = lambda_vals[i]
    try:
        non_private_problem.solve(solver=cp.ECOS)
    except cp.error.SolverError:
        print("SolverError at lambda = ", lamb.value)
        break
    if theta.value is None:
        print("Didn't converge at lambda = ", lamb.value)
        break
    train_error = error((X_train @ theta).value, y_train)
    test_error = error((X_test @ theta).value, y_test)
    train_loss = non_private_loss(X_train, y_train, theta.value, lamb.value)
```

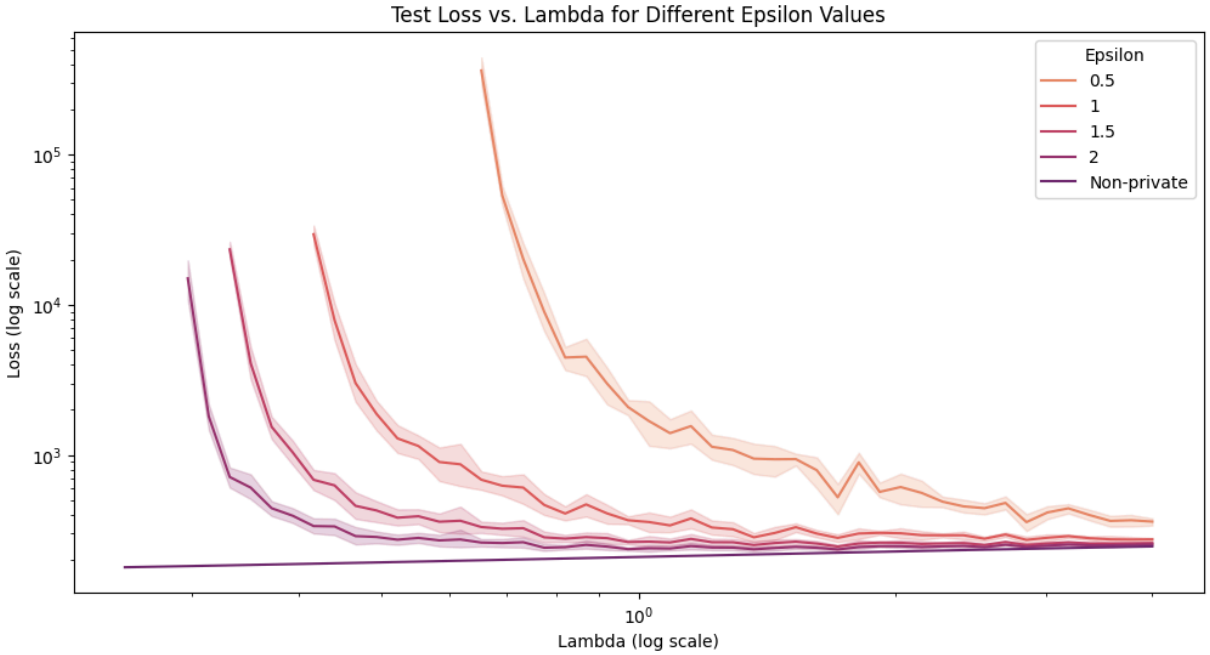
```
    test_loss = non_private_loss(X_test, y_test, theta.value, lamb.value)

    data.append([epsilon, lamb.value, train_error, train_loss, 'train'])
    data.append([epsilon, lamb.value, test_error, test_loss, 'test'])
```

In [12]:
```
df = pd.DataFrame(data, columns=['Epsilon', 'Lambda', 'Error', 'Loss', 'Type
# df = df.sort_values('Epsilon', key=lambda x: x.map({0.5: 1, 1: 2, 1.5: 3,
df = df.sort_values('Epsilon', key=lambda x: x.map({ep: i for i, ep in enume
df['Epsilon'] = df['Epsilon'].astype(str)
tests_only = df[df['Type'] == 'test']
```

In [13]:
```
plt.figure(figsize=(12, 6))
plot = sns.lineplot(x='Lambda', y='Error', hue='Epsilon', data=tests_only, e
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Lambda (log scale)')
plt.ylabel('Error (log scale)')
plt.title('Test Error vs. Lambda for Different Epsilon Values')
plt.show()
```

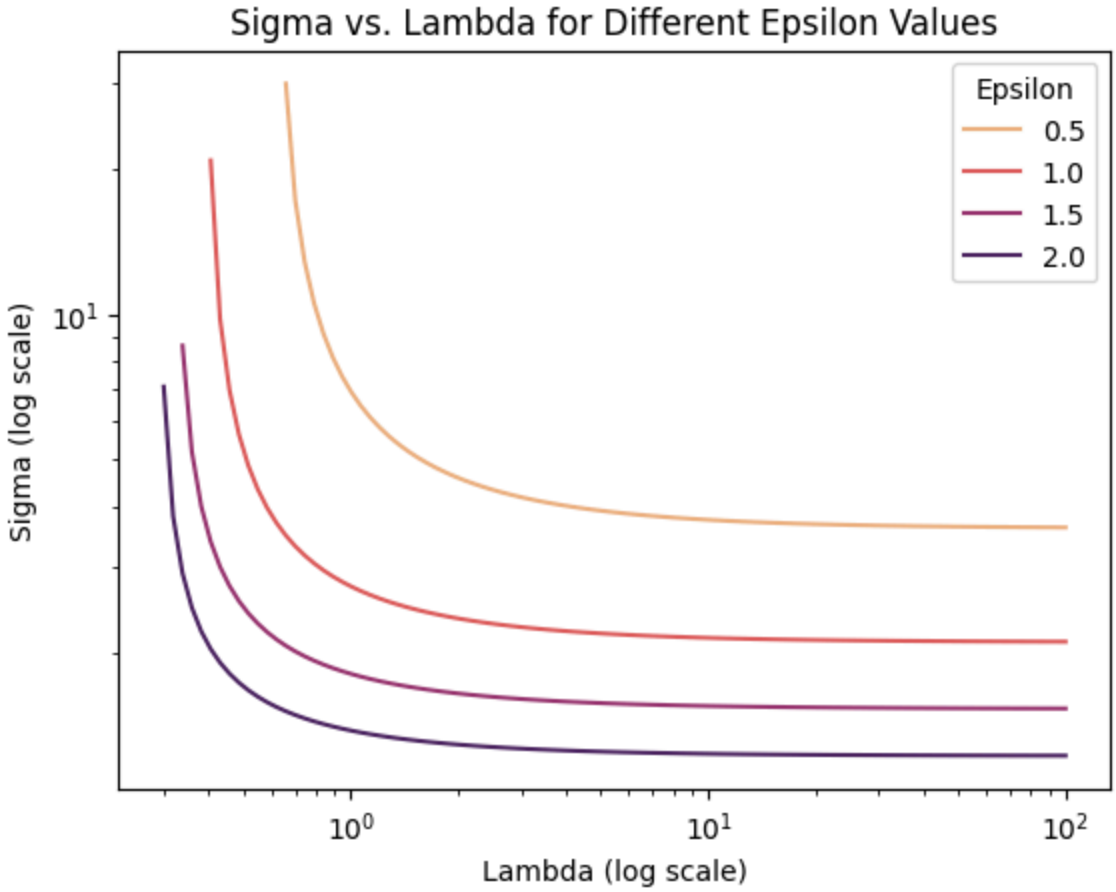


In [14]:
```
plt.figure(figsize=(12, 6))
plot = sns.lineplot(x='Lambda', y='Loss', hue='Epsilon', data=tests_only, er
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Lambda (log scale)')
plt.ylabel('Loss (log scale)')
plt.title('Test Loss vs. Lambda for Different Epsilon Values')
plt.show()
```



In [15]:
```
lams = np.logspace(np.log10(beta + 1e-4), np.log10(100), 100)
eps = [0.5, 1, 1.5, 2]
delt = 1e-2
sigs = [[ep, lam, sigma_(ep, delt, beta, lam, L)] for ep in eps for lam in l

lines = pd.DataFrame(sigs, columns=['Epsilon', 'Lambda', 'Sigma'])

sns.lineplot(x='Lambda', y='Sigma', hue='Epsilon', data=lines, palette='flar
plt.yscale('log')
plt.xscale('log')
plt.xlabel('Lambda (log scale)')
plt.ylabel('Sigma (log scale)')
plt.title('Sigma vs. Lambda for Different Epsilon Values')
plt.show()
```

Objective Perturbation



FTRL

In [1]:
```
import numpy as np
import pandas as pd
from os import path
import matplotlib.pyplot as plt
import seaborn as sns
from typing import Callable, Tuple, Optional
from numpy.linalg import norm

from sklearn.linear_model import LogisticRegression
from sklearn import preprocessing

from autodp import mechanism_zoo, calibrator_zoo
from autodp.transformer_zoo import ComposeGaussian
```

In [2]:
```
# Set the Seaborn style
#sns.set_theme()
# Get the "flare" palette
color_palette = sns.color_palette("flare")
```

In [3]:
```
df = pd.read_csv(path.join('OPS-SAT-AD-main', 'data', 'dataset.csv'), index_

#convert categorical column to multiple binary
df=pd.get_dummies(df,columns=["channel"])

X = df[[x for x in df.columns if x!="anomaly"]]
y = df["anomaly"].to_numpy()

# First normalize the individual data points
dim = X.shape[1]
n = X.shape[0]

# Rescaling the feature vectors by their natural ranges (independent to the
# X = X @ np.diag(1./np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 10,1000,10000,1

# the following bounds are chosen independent to the data
x_bound = 1
y_bound = 1

# Preprocess the feature vector such that the norm is fixed at 5
X = x_bound*preprocessing.normalize(X, norm='l2')
```

In [16]:
```
class DPFTRLPrivacyEngine:
    """Privacy engine for DP-FTRL with tree restart"""
    def __init__(self, total_epsilon: float, epochs: int, delta: float, heig
        self.total_epsilon = total_epsilon
        self.epochs = epochs
        self.delta = delta
        self.height = height

        # Allocate epsilon budget equally across epochs due to tree restart
        self.epsilon_per_epoch = total_epsilon / epochs
```

FTRL

```
        # Initialize calibrator
        self.calibrator = calibrator_zoo.ana_gaussian_calibrator()

    def get_sigma(self, l2_clip: float) -> float:
        """Get optimal sigma for noise calibration"""
        mechanism = self.calibrator.calibrate(
            mech_class=mechanism_zoo.ExactGaussianMechanism,
            eps=self.epsilon_per_epoch,
            delta=self.delta/self.epochs
        )
        return mechanism.params['sigma'] * l2_clip * np.sqrt(self.height)

class DPTreeAggregator:
    """Binary tree structure for private gradient aggregation"""
    def __init__(self, num_steps: int, dim: int, sigma: float):
        self.height = int(np.ceil(np.log2(num_steps))) + 1
        self.dim = dim
        self.tree = np.zeros((2**self.height - 1, dim))
        self.sigma = sigma
        self.noise = np.zeros((2**self.height - 1, dim))

    def add_gradient(self, step: int, gradient: np.ndarray) -> None:
        idx = 2**(self.height - 1) + step
        self.tree[idx] = gradient

        while idx > 0:
            parent = (idx - 1) // 2
            if idx % 2 == 1:
                self.tree[parent] = self.tree[parent - 1] + self.tree[idx]
            else:
                self.tree[parent] = self.tree[idx]
            idx = parent

    def get_noisy_sum(self, step: int) -> np.ndarray:
        total = np.zeros(self.dim)
        idx = 2**(self.height - 1) + step
        # print(self.sigma)
        while idx > 0:
            parent = (idx - 1) // 2
            if idx % 2 == 1:
                if (self.noise[parent] == np.zeros(self.dim)).any():
                    self.noise[parent] = np.random.normal(0, self.sigma, sel
                total += self.tree[idx] + self.noise[parent]
            idx = parent
        return total

class DPFTRLM:
    def __init__(
        self,
        dim: int,
        num_examples: int,
        batch_size: int,
```

FTRL

```
        learning_rate: float,
        momentum: float,
        l2_norm_clip: float,
        lambda_reg: float,
        target_epsilon: float,
        target_delta: float,
        epochs: int,
        loss_grad_fn: Callable
    ):
        self.dim = dim
        self.num_examples = num_examples
        self.batch_size = batch_size
        self.lr = learning_rate
        self.momentum = momentum
        self.l2_clip = l2_norm_clip
        self.lambda_reg = lambda_reg
        self.epochs = epochs
        self.loss_grad_fn = loss_grad_fn

        # Initialize privacy engine
        self.privacy_engine = DPFTRLPrivacyEngine(
            total_epsilon=target_epsilon,
            epochs=epochs,
            delta=target_delta,
            height= int(np.ceil(self.num_examples / self.batch_size))
        )

        # Initialize momentum buffer and initial point
        self.velocity = np.zeros(dim)
        self.theta_0 = np.zeros(dim)

    def get_eps(self):
        return self.privacy_engine.total_epsilon

    def train_epoch(self, X: np.ndarray, y: np.ndarray, theta: np.ndarray) -
        num_batches = int(np.ceil(self.num_examples / self.batch_size))

        # Initialize tree for this epoch with calibrated noise
        tree = DPTreeAggregator(
            num_steps=num_batches,
            dim=self.dim,
            sigma=self.privacy_engine.get_sigma(self.l2_clip)
        )

        perm = np.random.permutation(self.num_examples)
        X, y = X[perm], y[perm]

        for batch in range(num_batches):
            start_idx = batch * self.batch_size
            end_idx = min(start_idx + self.batch_size, self.num_examples)
            X_batch = X[start_idx:end_idx]
            y_batch = y[start_idx:end_idx]
```

FTRL

```
            grads = self.loss_grad_fn(X_batch, y_batch, theta)
            grad_norms = np.linalg.norm(grads, axis=1)
            grad_norms = np.maximum(grad_norms, 1e-12)  # Prevent division b
            scaling = np.minimum(1, self.l2_clip / grad_norms)
            grads = grads * scaling[:, np.newaxis]

            tree.add_gradient(batch, np.mean(grads, axis=0))

            noisy_grads = tree.get_noisy_sum(batch)

            # Update with momentum and regularization
            self.velocity = self.momentum * self.velocity + noisy_grads
            reg_grad = self.lambda_reg * (theta - self.theta_0)
            theta = theta - self.lr * (self.velocity + reg_grad)

        # With tree restart, privacy spent per epoch is fixed
        privacy_spent = self.privacy_engine.epsilon_per_epoch
        self.privacy_engine.total_epsilon -= privacy_spent
        error = err(X, y, theta)

        return theta, privacy_spent, error

def err(X, y, theta):
    return np.sum((X @theta > 0) != y) / X.shape[0]

def err_yhat(y, yhat):
    return np.sum((yhat != y)) / len(y)

def logistic_loss_grad(X: np.ndarray, y: np.ndarray, theta: np.ndarray) -> r
    """Compute per-example gradients for logistic regression

    Args:
        X: Input features of shape (batch_size, dim)
        y: Labels of shape (batch_size,)
        theta: Model parameters of shape (dim,)

    Returns:
        Gradients of shape (batch_size, dim)
    """
    # Compute predictions
    z = X @ theta  # Shape: (batch_size,)
    sigmoid = 1 / (1 + np.exp(-z))  # Shape: (batch_size,)

    # Compute per-example gradients
    grads = X * (sigmoid - y)[:, np.newaxis]  # Shape: (batch_size, dim)
    return grads
```

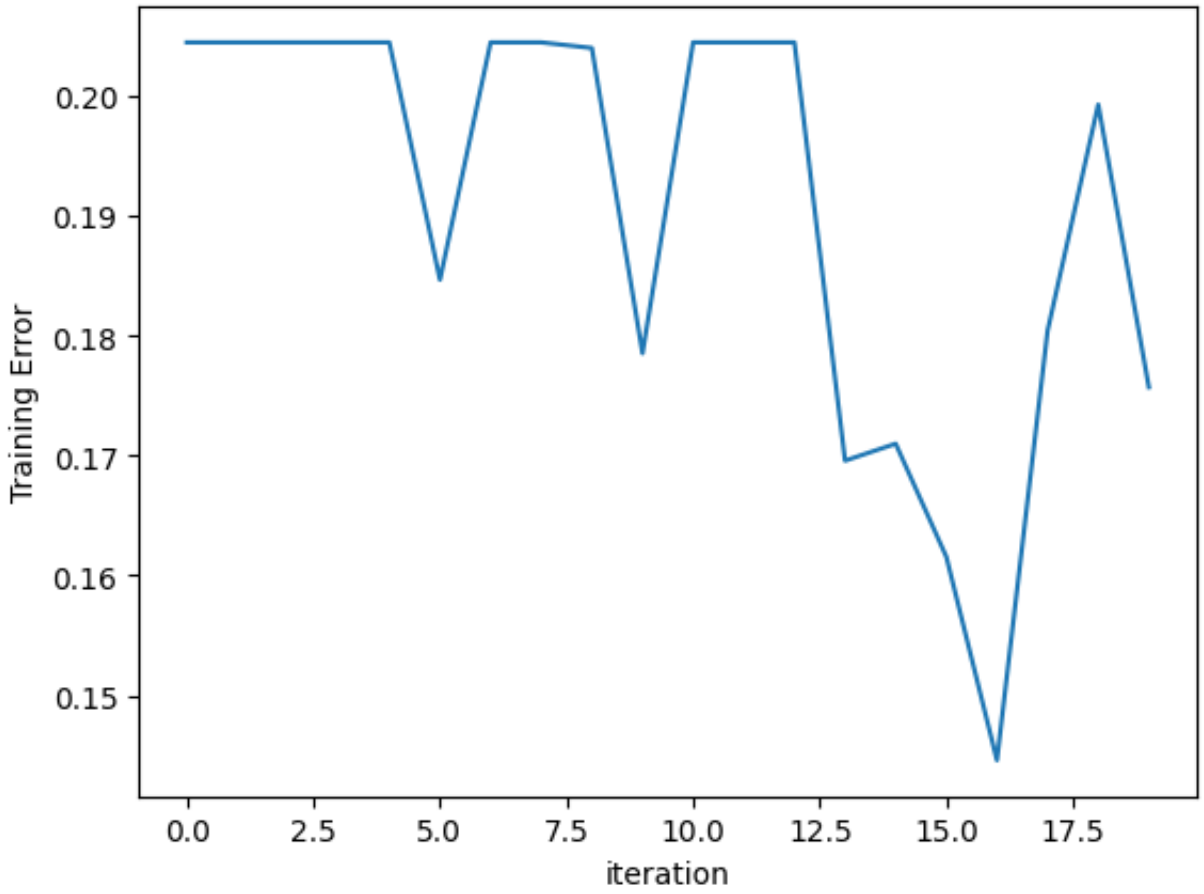
FTRL

In [ ]:
```python
# LARGE BUDGET IS ESSENTIALLY NON PRIVATE FTRL BASELINE

lamb = .01
learning_rate = 1 / (lamb * X.shape[0])

# Initialize optimizer
optimizer = DPFTRLM(
    dim=X.shape[1],
    num_examples=X.shape[0],
    batch_size=100,
    learning_rate=learning_rate, # 1 / lambda*num_examples
    momentum=0.9,
    l2_norm_clip=1.0,
    lambda_reg=lamb, # Add regularization strength
    target_epsilon=10000,
    target_delta=1e-5,
    epochs=20,
    loss_grad_fn=logistic_loss_grad
)

# Training loop
theta = np.zeros(optimizer.dim)
errs = []
for epoch in range(optimizer.epochs):
    theta, eps, error = optimizer.train_epoch(X, y, theta)
    errs.append(error)
    # if epoch % 10 == 0:
    #     print(f"Epoch {epoch}, ε = {eps}")

print("remaining budget", optimizer.get_eps())

pd.Series(errs).plot()
plt.xlabel("iteration")
plt.ylabel("Training Error")
```

```
remaining budget 0.0
```

Out[ ]:
```
Text(0, 0.5, 'Training Error')
```

FTRL


Training Error
0.20
0.19
0.18
0.17
0.16
0.15
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
iteration

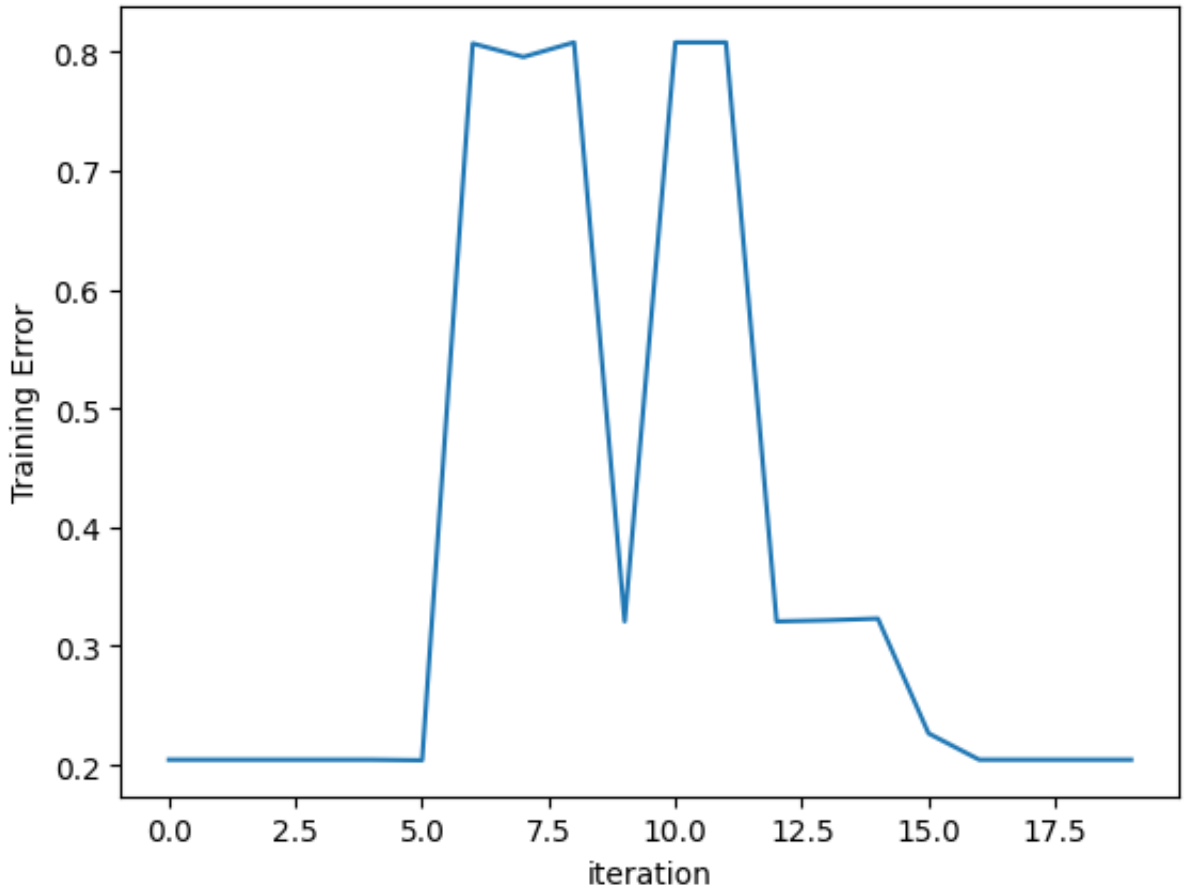
FTRL

In [22]:
```
lamb = .01
learning_rate = 1 / (lamb * X.shape[0])

# Initialize optimizer
optimizer = DPFTRLM(
    dim=X.shape[1],
    num_examples=X.shape[0],
    batch_size=100,
    learning_rate=learning_rate, # 1 / lambda*num_examples
    momentum=0.9,
    l2_norm_clip=1.0,
    lambda_reg=lamb, # Add regularization strength
    target_epsilon=1,
    target_delta=1e-5,
    epochs=20,
    loss_grad_fn=logistic_loss_grad
)

# Training loop
theta = np.zeros(optimizer.dim)
errs = []
for epoch in range(optimizer.epochs):
    theta, eps, error = optimizer.train_epoch(X, y, theta)
    errs.append(error)
    # if epoch % 10 == 0:
    #     print(f"Epoch {epoch}, ε = {eps}")

print("remaining budget", optimizer.get_eps())

pd.Series(errs).plot()
plt.xlabel("iteration")
plt.ylabel("Training Error")
```

```
remaining budget -3.191891195797325e-16
/var/folders/31/df158hy936q0z2m_dr_493pm0000gn/T/ipykernel_52402/183342307
9.py:158: RuntimeWarning: overflow encountered in exp
  sigmoid = 1 / (1 + np.exp(-z))  # Shape: (batch_size,)
```

Out[22]:
```
Text(0, 0.5, 'Training Error')
```

FTRL



In [5]:
```
clf = LogisticRegression(random_state=0,fit_intercept=False).fit(X, y)
yhat = clf.predict(X)

err_nonprivate = err_yhat(y, yhat)
err_trivial = min(np.mean(y), 1-np.mean(y) )

# Nonprivate baseline
print('Nonprivate error rate is', err_yhat(y, yhat))

print('Trivial error rate is', err_trivial)
```

```
Nonprivate error rate is 0.13141780499293454
Trivial error rate is 0.20442769665567592
```

In [7]:
```
def run_tune_hyperparameters(X, y, dim, num_examples, l2_clip,
                             target_epsilon, target_delta, epochs):
    # Grid for hyperparameters
    momentums = [0.9, 0.97, 0.85, 0.7]
    batch_sizes = [100, 200, 150]
    lambdas = [0.001, 0.01, 0.1, 1.0]

    best_error = float('inf')
    best_params = None
    results = []

    for m in momentums:
        for lamb in lambdas:
```

FTRL

```
            adjusted_lr = 1 / (lamb * num_examples)
            for bs in batch_sizes:
                optimizer = DPFTRLM(
                    dim=dim,
                    num_examples=num_examples,
                    batch_size=bs,
                    learning_rate=adjusted_lr,
                    momentum=m,
                    l2_norm_clip=l2_clip,
                    lambda_reg=lamb,
                    target_epsilon=target_epsilon,
                    target_delta=target_delta,
                    epochs=epochs,
                    loss_grad_fn=logistic_loss_grad
                )

                # Training loop
                theta = np.zeros(optimizer.dim)
                final_error = float('inf')

                for epoch in range(optimizer.epochs):
                    theta, eps, error = optimizer.train_epoch(X, y, theta)
                    final_error = error

                results.append({
                    'momentum':m,
                    'batch_size': bs,
                    'lambda': lamb,
                    'final_error': final_error
                })

                if final_error < best_error:
                    best_error = final_error
                    best_params = {'batch_size': bs, 'momentum': m, 'lambda'

                # print(f"momentum: {m:.2f}, batch_size: {bs}, lambda: {lamb

    print("\nBest parameters:")
    print(f"Momentum: {best_params['momentum']:.2f}")
    print(f"batch_size: {best_params['batch_size']}")
    print(f"Lambda: {best_params['lambda']:.6f}")
    print(f"Best error: {best_error:.4f}")

    return best_params, results

# Usage example
best_params, results = run_tune_hyperparameters(
    X=X,
    y=y,
    dim=X.shape[1],
    num_examples=X.shape[0],
    #batch_size=100,
    #momentum=0.9,
```

FTRL

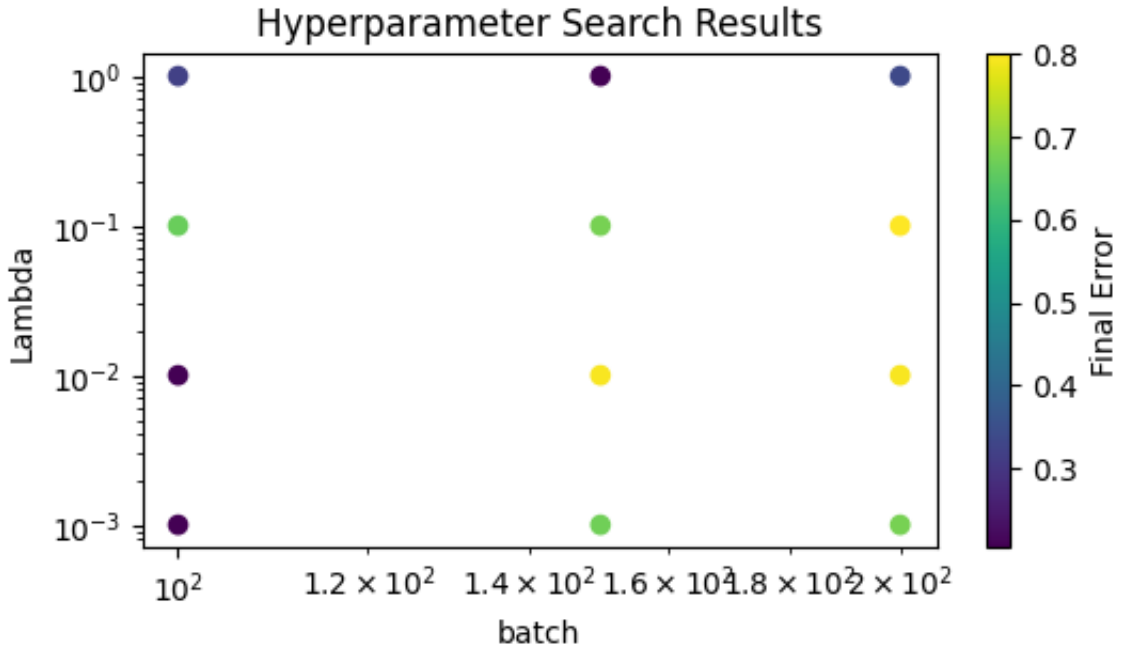
```
    l2_clip=1.0,
    target_epsilon=100,
    target_delta=1e-5,
    epochs=50
)
```

```
/var/folders/31/df158hy936q0z2m_dr_493pm0000gn/T/ipykernel_52402/745166385.
py:158: RuntimeWarning: overflow encountered in exp
  sigmoid = 1 / (1 + np.exp(-z))  # Shape: (batch_size,)
Best parameters:
Momentum: 0.70
batch_size: 100
Lambda: 1.000000
Best error: 0.1842
```

In [8]:
```
df_results = pd.DataFrame(results).groupby(["batch_size", "lambda"]).max().r
plt.figure(figsize=(6, 3))
plt.scatter(df_results['batch_size'], df_results['lambda'],
           c=df_results['final_error'], cmap='viridis')
plt.colorbar(label='Final Error')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('batch')
plt.ylabel('Lambda')
plt.title('Hyperparameter Search Results')
print(df_results["momentum"])
plt.show()
```

```
0     0.97
1     0.97
2     0.97
3     0.97
4     0.97
5     0.97
6     0.97
7     0.97
8     0.97
9     0.97
10    0.97
11    0.97
Name: momentum, dtype: float64
```

FTRL

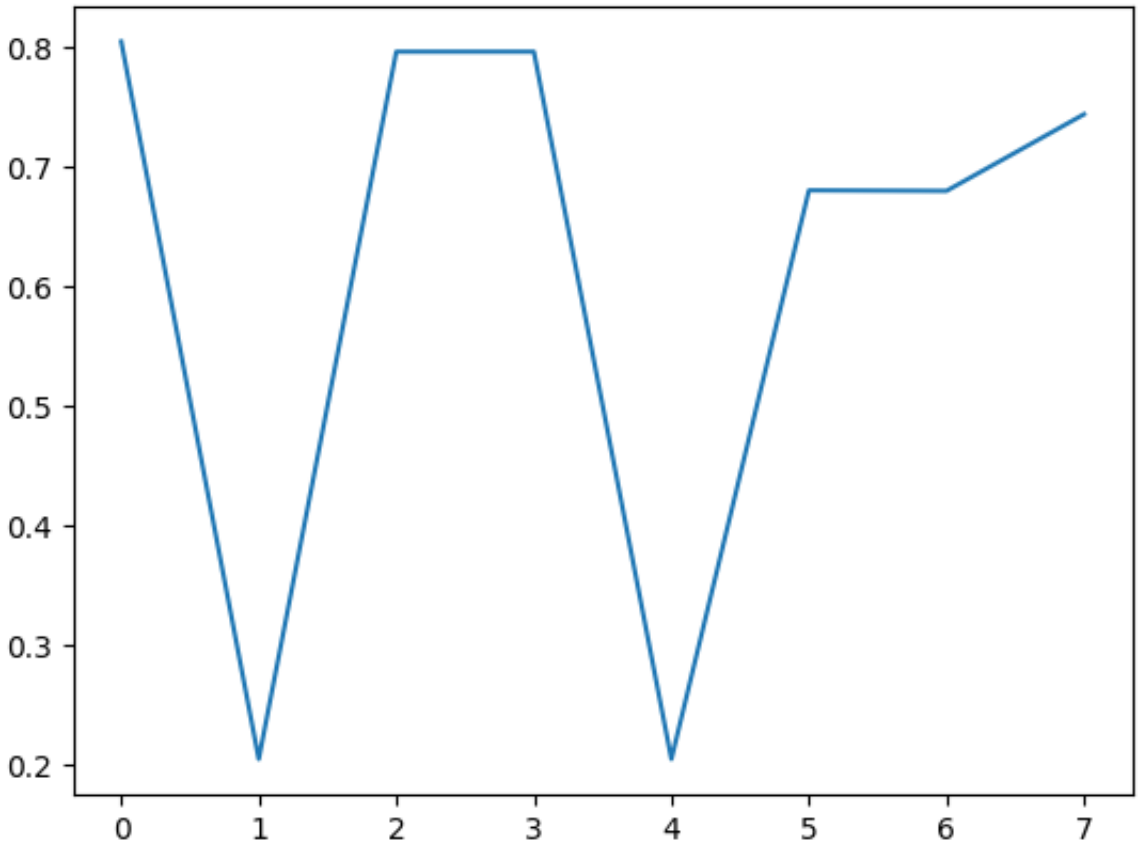


In [9]:

```
best_errs = []
lamb = .01
epss = [0.005, 0.01, 0.1, 0.25, 0.5, 1, 1.5, 2]
for test_eps in epss:
    # Initialize optimizer
    optimizer = DPFTRLM(
        dim=X.shape[1],
        num_examples=X.shape[0],
        batch_size=100,
        learning_rate=1 / (lamb * X.shape[0]), # 1 / lambda?
        momentum=0.9,
        l2_norm_clip=1.0,
        lambda_reg=lamb, # Add regularization strength
        target_epsilon=test_eps,
        target_delta=1e-5,
        epochs=100,
        loss_grad_fn=logistic_loss_grad
    )

    # Training loop
    theta = np.zeros(optimizer.dim)
    for epoch in range(optimizer.epochs):
        theta, eps, error = optimizer.train_epoch(X, y, theta)
        #if epoch % 10 == 0:
        #    print(f"Epoch {epoch}, ε = {eps}")

    if abs(optimizer.get_eps()) > .01:
        print("eps: ",optimizer.get_eps())

    best_errs.append(error)

pd.Series(best_errs).plot();
```

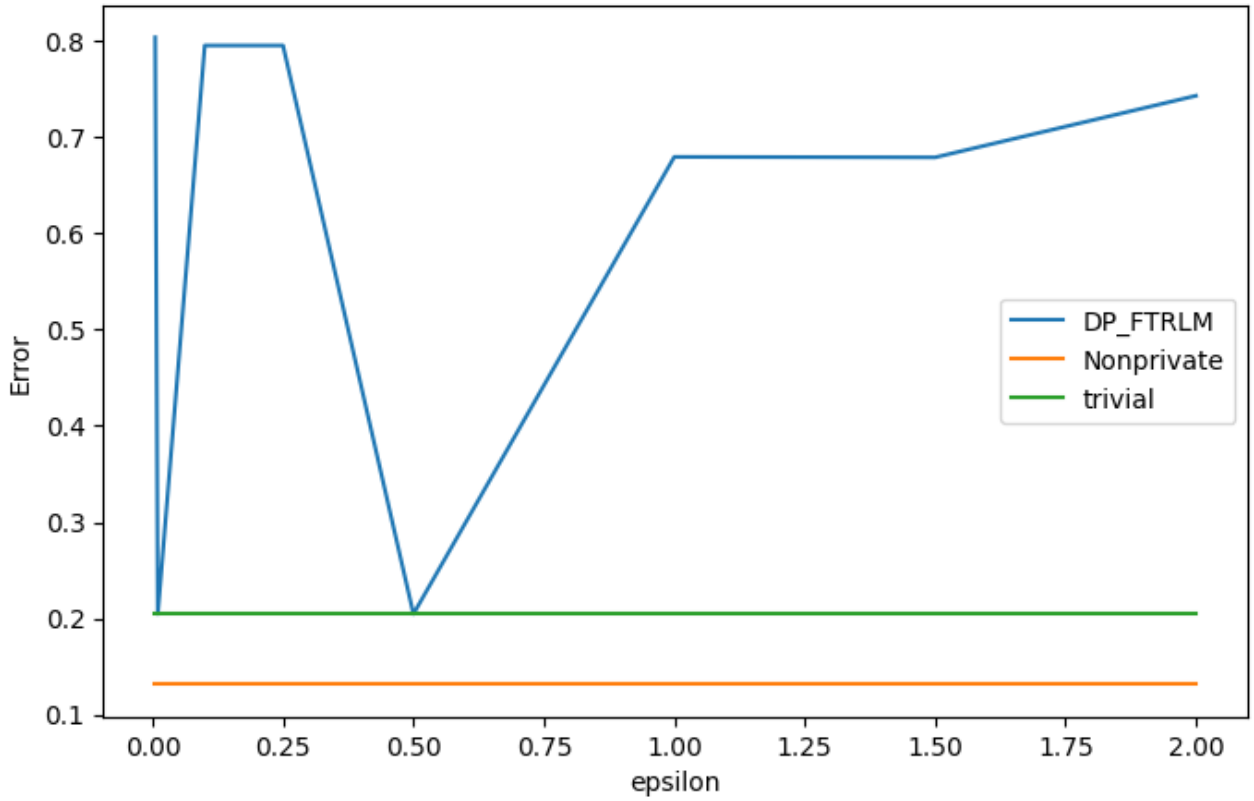
FTRL

```
/var/folders/31/df158hy936q0z2m_dr_493pm0000gn/T/ipykernel_52402/745166385.
py:158: RuntimeWarning: overflow encountered in exp
  sigmoid = 1 / (1 + np.exp(-z))  # Shape: (batch_size,)
```



In [12]:
```
plt.figure(figsize=(8, 5))


plt.plot(epss,np.array(best_errs))#,color_palette[0])
plt.plot(epss,err_nonprivate*np.ones_like(epss))#,color_palette[2])
plt.plot(epss,err_trivial*np.ones_like(epss))#,color_palette[4])

plt.legend(['DP_FTRLM','Nonprivate','trivial'])
plt.xlabel('epsilon')
plt.ylabel('Error')
plt.show()
```

FTRL


0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
Error
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
epsilon
DP_FTRLM
Nonprivate
trivial

In [ ]:

Output Perturbation

In [5]:
```
from sklearn.preprocessing import LabelEncoder
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
import numpy as np
from scipy.optimize import minimize
from scipy.stats import norm
import matplotlib.pyplot as plt
```

In [6]:
```
import os

df = pd.read_csv('data/dataset.csv')
```

In [7]:
```
#convert categorical column to multiple binary
df=pd.get_dummies(df,columns=["channel"])
```

In [8]:
```
from sklearn.linear_model import LogisticRegression
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import numpy as np

dataset = df
print('This is a regression dataset.')
print('Features are: ', "".join(dataset.columns))
print('The label is: ', "anomaly")
print('The shape of the data matrix iss', dataset.shape)

# Let's extract the relevant information from the sklearn dataset object
X = dataset[[x for x in dataset.columns if x!="anomaly"]]
y = dataset["anomaly"]

# -------------- Uncomment the following to test size = 0.9 when debugging you code----
#
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.9, random_state

# X = X_train
# y = y_train
# -------------------- But please submit your code without taking a random subset -----

# First normalize the individual data points

dim = X.shape[1]
n = X.shape[0]



# This is very important for DP linear regression.
# Also please make sure that the preprocessing does not use dataset for training,
# e.g., the standard z-score cannot be used.



# Rescaling the feature vectors by their natural ranges (independent to the data)
#re-scale correctly
```

Output Perturbation

```
#X = X @ np.diag(1./np.array([10,50,100,40,40000,1000,50,100]))
# This is to ensure that each feature is of the similar scale

# the following bounds are chosen independent to the data
x_bound = 1
y_bound = 1

# Preprocess the feature vector such that the norm is fixed at 5
X = x_bound*preprocessing.normalize(X, norm='l2')



def CE(score,y):
    # numerically efficient vectorized implementation of CE loss
    log_phat = np.zeros_like(score)
    log_one_minus_phat = np.zeros_like(score)
    mask = score > 0
    log_phat[mask] = - np.log( 1 + np.exp(-score[mask]))
    log_phat[~mask] = score[~mask] - np.log( 1 + np.exp(score[~mask]))
    log_one_minus_phat[mask] = -score[mask] - np.log( 1 + np.exp(-score[mask]))
    log_one_minus_phat[~mask] = - np.log( 1 + np.exp(score[~mask]))

    return -y*log_phat-(1-y)*log_one_minus_phat




def loss(theta):
    return np.sum(CE(X@theta,y))/n

def err(theta):
    return np.sum((X@theta > 0) != y) / n

def err_yhat(yhat):
    return np.sum((yhat != y)) / n


clf = LogisticRegression(penalty = 'l2', random_state=0, fit_intercept=False).fit(X, y)
yhat = clf.predict(X)

err_nonprivate = err_yhat(yhat)
err_trivial = min(np.mean(y), 1-np.mean(y))
theta_values = clf.coef_[0]  # Get the coefficients

# Print results
print('Nonprivate error rate is', err_nonprivate)
print('Trivial error rate is', err_trivial)
print('Theta values:', theta_values)
```

```
This is a regression dataset.
Features are:  segmentanomalytrainsamplingdurationlenmeanvarstdkurtosisskewn_peakssmooth
10_n_peakssmooth20_n_peaksdiff_peaksdiff2_peaksdiff_vardiff2_vargaps_squaredlen_weighted
var_div_durationvar_div_lenchannel_CADC0872channel_CADC0873channel_CADC0874channel_CADC0
884channel_CADC0886channel_CADC0888channel_CADC0890channel_CADC0892channel_CADC0894
The label is:  anomaly
The shape of the data matrix iss (2123, 31)
Nonprivate error rate is 0.14554875176636833
Trivial error rate is 0.20442769665567592
Theta values: [-1.68573475e+00 -2.07321845e-02 -1.48530878e-01  1.74711557e-01
  4.07115718e+00 -1.49419539e-02 -5.50235994e-03 -1.33873828e-02
```

Output Perturbation

```
 1.50754986e-01 -4.16416483e-03  1.55714063e-01  8.52301951e-02
 1.22851992e-02 -6.37276540e+00 -7.25971729e+00  2.70195977e-04
 9.35128119e-04 -1.21618526e+00 -5.43319048e-02 -1.89151922e-05
-4.11073569e-05 -3.30886539e-02  1.78041624e-02  6.44087001e-03
-1.59006706e-02  6.04395975e-04  1.26815141e-02  4.68218448e-03
-1.37010005e-02 -3.74948699e-03]
```

In [9]:

```
from autodp.autodp_core import Mechanism
from autodp.mechanism_zoo import GaussianMechanism
from autodp.calibrator_zoo import eps_delta_calibrator
from autodp.mechanism_zoo import ExactGaussianMechanism, PureDP_Mechanism




def CE(score, y):
    """Compute cross-entropy loss"""
    log_phat = np.zeros_like(score)
    log_one_minus_phat = np.zeros_like(score)
    mask = score > 0
    log_phat[mask] = - np.log(1 + np.exp(-score[mask]))
    log_phat[~mask] = score[~mask] - np.log(1 + np.exp(score[~mask]))
    log_one_minus_phat[mask] = -score[mask] - np.log(1 + np.exp(-score[mask]))
    log_one_minus_phat[~mask] = - np.log(1 + np.exp(score[~mask]))
    return -y*log_phat-(1-y)*log_one_minus_phat

def loss(theta, X, y, lambda_reg):
    """Compute total loss with L2 regularization"""
    # L2 regularization directly to the loss function.
    return np.sum(CE(X@theta, y))/len(y) + (lambda_reg/2) * np.sum(theta**2)

def err(theta, X, y):
    """Compute classification error"""
    return np.sum((X@theta > 0) != y) / len(y)


#This function trains the model and adds calibrated noise to ensure differential privac
#Lambda needs to grow with n for the sensitivity to be meaningful.

def train_private_logistic_regression(X, y, lambda_reg, epsilon, delta = 1e-6, niter=10
    """Train logistic regression with output perturbation"""
    n, d = X.shape
    theta = np.zeros(d)

    X = x_bound*preprocessing.normalize(X, norm='l2')

    # Train standard logistic regression
    clf = LogisticRegression(random_state=0, fit_intercept=False).fit(X, y)
    yhat = clf.predict(X)

    err_nonprivate = err_yhat(yhat)
    err_trivial = min(np.mean(y), 1-np.mean(y))
    theta_values = clf.coef_[0]  # Get the coefficients

    calibrate = eps_delta_calibrator()
    eps = epsilon
    delta = delta

    mech1 = calibrate(ExactGaussianMechanism,eps,delta,[0,100],name='GM1')
```

Output Perturbation

```
    sigma = mech1.params['sigma']


    noise = np.random.normal(0, sigma/lambda_reg, size=d)
    private_theta = theta_values + noise


    return private_theta, err(private_theta, X, y)
```

In [10]:

```
def calculate_lambda(n, epsilon, d, L, theta_star_norm):
    delta = 1e-6  # Fixed privacy parameter
    numerator = (d**(1/3)) * (delta**(1/3)) * (L**(2/3)) * \
                (theta_star_norm**(4/3)) * (np.log(1/delta)**(1/3))
    denominator = (n**(2/3)) * (epsilon**(2/3))
    return n*(numerator / denominator)


epsilon = 1
n = X.shape[0]
d = X.shape[1]
L = 1.0
theta_star_norm = 5.0

lambda_value = calculate_lambda(n, epsilon, d, L, theta_star_norm)

lambda_value
```

Out[10]:

```
8.19293297097756
```

In [11]:

```
train_private_logistic_regression(X, y, lambda_reg = lambda_value, epsilon =1)
```

Out[11]:

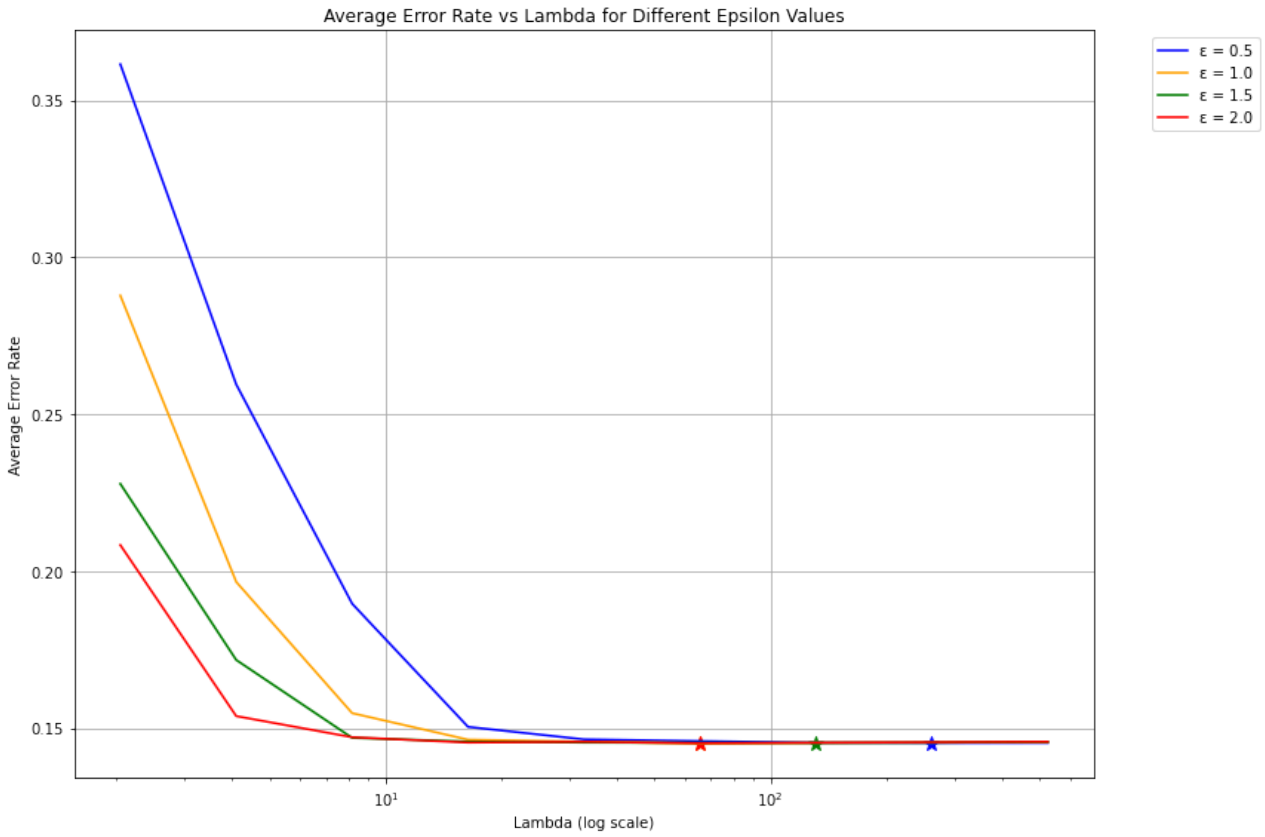
```
(array([-1.79738829,  0.96021636,  0.02054781,  0.27182299,  3.47803867,
        -0.13404738,  0.54789644,  0.91708246,  0.34900632, -0.58632083,
        -0.54181829, -0.4973831 ,  0.9486871 , -7.18697087, -6.88908249,
        -0.62072169,  0.16299166, -0.93381767,  0.48671743,  0.78404373,
         0.59465405, -0.55755576, -0.35535589,  0.33181575,  0.21961103,
         0.30669977, -0.48881642,  0.07972172,  0.04005354,  0.16193117]),
 0.14413565708902495)
```

In [12]:

```
epsilons = [0.5, 1.0, 1.5, 2.0]
base_lambda = lambda_value
multipliers = [1/4, 1/2, 1, 2, 4,8,16,32,64]
lambdas = [base_lambda * m for m in multipliers]
delta = 1e-6
n_runs = 100

# Create the plot
plt.figure(figsize=(12, 8))
colors = ['blue', 'orange', 'green', 'red']
epsilon_colors = dict(zip(epsilons, colors))
optimal_lambdas = []
```

Output Perturbation

```
# Plot a line for each epsilon value
for epsilon in epsilons:
    # Initialize array to store errors for each lambda
    all_errors = np.zeros((n_runs, len(lambdas)))

    for run in range(n_runs):
        for i, lambda_reg in enumerate(lambdas):
            _, error = train_private_logistic_regression(X, y, lambda_reg, epsilon, del
            all_errors[run, i] = error

    # Calculate average errors across runs
    avg_errors = np.mean(all_errors, axis=0)

    # Plot average line
    plt.plot(lambdas, avg_errors, color=epsilon_colors[epsilon], label=f'ε = {epsilon}'

    # Find and plot optimal point
    min_error_idx = np.argmin(avg_errors)
    optimal_lambda = lambdas[min_error_idx]
    optimal_lambdas.append(optimal_lambda)
    plt.scatter(optimal_lambda, avg_errors[min_error_idx], marker='*', s=100, color=eps

plt.xscale('log')
plt.xlabel('Lambda (log scale)')
plt.ylabel('Average Error Rate')
plt.title('Average Error Rate vs Lambda for Different Epsilon Values')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.tight_layout()
plt.show()
```



In [13]:

```
optimal_lambdas
```

Output Perturbation

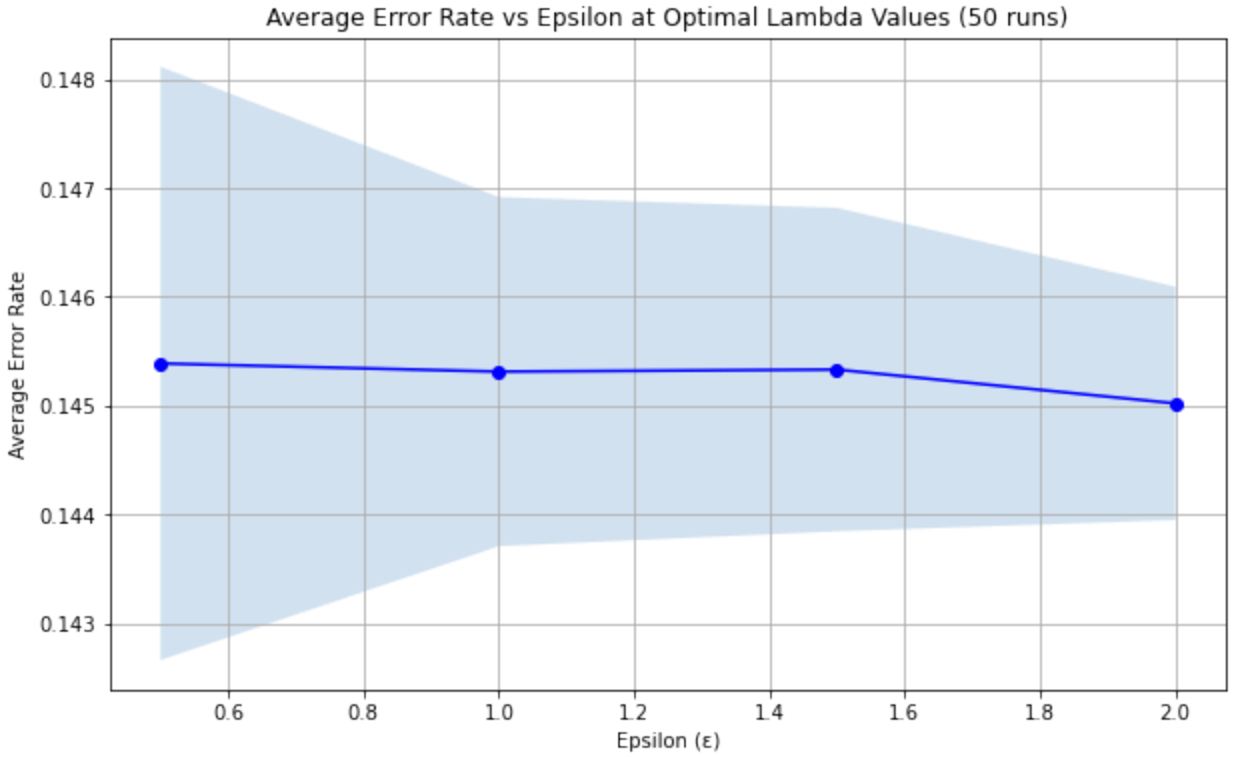
Out[13]:
```
[262.17385507128193, 65.54346376782048, 131.08692753564097, 65.54346376782048]
```

Now I am going to try and find the ideal lambda_reg I will set the epsilon at .1 and the niter to 1000 and try different lambda parameters and see how the error changes

In [19]:
```
epsilons = [0.5, 1.0, 1.5, 2.0]
delta = 1e-6
n_runs = 50
# Initialize storage for errors
all_errors = {eps: [] for eps in epsilons}

# Run multiple times for each epsilon with its corresponding optimal lambda
for epsilon, optimal_lambda in zip(epsilons, optimal_lambdas):
    for _ in range(n_runs):
        private_theta, error = train_private_logistic_regression(X, y, optimal_lambdas[
        all_errors[epsilon].append(error)

# Calculate average errors
avg_errors = [np.mean(all_errors[eps]) for eps in epsilons]
std_errors = [np.std(all_errors[eps]) for eps in epsilons]

# Create plot
plt.figure(figsize=(10, 6))
plt.plot(epsilons, avg_errors, 'b-o')
plt.fill_between(epsilons,
                 [avg - std for avg, std in zip(avg_errors, std_errors)],
                 [avg + std for avg, std in zip(avg_errors, std_errors)],
                 alpha=0.2)
plt.xlabel('Epsilon (ε)')
plt.ylabel('Average Error Rate')
plt.title(f'Average Error Rate vs Epsilon at Optimal Lambda Values ({n_runs} runs)')
plt.grid(True)

# Print results
print("\nResults averaged over 50 runs:")
for eps, avg, std in zip(epsilons, avg_errors, std_errors):
    print(f"ε = {eps:.1f}: Error = {avg:.4f} ± {std:.4f}")

plt.show()
```

```
Results averaged over 50 runs:
ε = 0.5: Error = 0.1454 ± 0.0027
ε = 1.0: Error = 0.1453 ± 0.0016
ε = 1.5: Error = 0.1453 ± 0.0015
ε = 2.0: Error = 0.1450 ± 0.0011
```

Output Perturbation



In [ ]: