CS 189 Introduction to Machine Learning
Spring 2024 Jonathan Shewchuk

HW4

Due: Friday, March 8 at 11:59 PM PST

- Homework 4 consists of coding assignments and math problems.
- We prefer that you typeset your answers using LaTeX or other word processing software. If you haven't yet learned LaTeX, one of the crown jewels of computer science, now is a good time! Neatly handwritten and scanned solutions will also be accepted.
- In all of the questions, **show your work**, not just the final answer.
- **We will not provide points back with respect to homework submission errors.** This includes, but is not limited to: 1) not assigning pages to problems; 2) not including code in the write-up appendix; 3) not including code in the designated code Gradescope assignment; 4) not including Kaggle scores; 5) submitting code that only partially works; 6). submitting late regrade requests. **Please carefully read and follow the HW submission guidelines/reminders for Pages 1, 2, and 10 of HW 4.**
- **Start early; you can submit models to Kaggle only twice a day!**

Deliverables:

1. Submit your predictions for the test sets to Kaggle as early as possible. Include your Kaggle scores in your write-up. The Kaggle competition for this assignment can be found at
   - WINE: https://www.kaggle.com/competitions/cs189-hw4-wine-spring-2024/
2. Write-up: Submit your solution in **PDF** format to "Homework 4 Write-Up" in Gradescope.
   - On the first page of your write-up, please list students with whom you collaborated
   - Start each question on a new page. If there are graphs, include those graphs on the same pages as the question write-up. DO NOT put them in an appendix. We need each solution to be self-contained on pages of its own.
   - **Only PDF uploads to Gradescope will be accepted**. You are encouraged use LaTeX or Word to typeset your solution. You may also scan a neatly handwritten solution to produce the PDF.
   - **Replicate all your code in an appendix**. Begin code for each coding question in a fresh page. Do not put code from multiple questions in the same page. When you upload this PDF on Gradescope, *make sure* that you assign the relevant pages of your code from appendix to correct questions.
   - While collaboration is encouraged, *everything* in your solution must be your (and only your) creation. Copying the answers or code of another student is strictly forbidden.

Furthermore, all external material (i.e., *anything* outside lectures and assigned readings, including figures and pictures) should be cited properly. We wish to remind you that consequences of academic misconduct are *particularly severe*!

3. Code: Submit your code as a .zip file to "Homework 4 Code".
   - **Set a seed for all pseudo-random numbers generated in your code.** This ensures your results are replicated when readers run your code. For example, you can seed `numpy` with `np.random.seed(189)`.
   - Include a README with your name, student ID, the values of random seed (above) you used, and instructions for running (and compiling, if appropriate) your code.
   - Do NOT provide any data files. Supply instructions on how to add data to your code.
   - Code requiring exorbitant memory or execution time might not be considered.
   - Code submitted here must match that in the PDF Write-up. The Kaggle score will not be accepted if the code provided a) does not compile or b) compiles but does not produce the file submitted to Kaggle.

Notation: In this assignment we use the following conventions.

- Symbol "defined equal to" ($\triangleq$) *defines* the quantity to its left to be the expression to its right and is equivalent to :=.
- Scalars are lowercase non-bold: x, u_1, α_i. Matrices are uppercase alphabets: A, B_1, C_i. Vectors (column vectors) are in bold: $\mathbf{x}, \boldsymbol{\alpha_1}, \mathbf{X}, \mathbf{Y_j}$.
- $\|\mathbf{v}\|$ denotes the Euclidean norm (length) of vector $\mathbf{v}$: $\|\mathbf{v}\| \triangleq \sqrt{\mathbf{v} \cdot \mathbf{v}}$. $\|A\|$ denotes the (operator) norm of matrix A, the magnitude of its largest singular value: $\|A\| = \max_{\|v\|=1} \|Av\|$.
- $[n] \triangleq \{1, 2, 3, \ldots, n\}$. $\mathbf{1}$ and $\mathbf{0}$ denote the vectors with all-ones and all-zeros, respectively.

1 Honor Code

Declare and sign the following statement (Mac Preview, PDF Expert, and FoxIt PDF Reader, among others, have tools to let you sign a PDF file):

"I certify that all solutions are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted."

Signature: 

2 Logistic Regression with Newton's Method

Given examples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ and associated labels $y_1, y_2, \ldots, y_n \in \{0, 1\}$, the cost function for *unregularized* logistic regression is

$$J(\mathbf{w}) \triangleq -\sum_{i=1}^{n} \Big(y_i \ln s_i + (1 - y_i) \ln(1 - s_i) \Big)$$

where $s_i \triangleq s(\mathbf{x}_i \cdot \mathbf{w})$, $\mathbf{w} \in \mathbb{R}^d$ is a weight vector, and $s(\gamma) \triangleq 1/(1 + e^{-\gamma})$ is the logistic function.

Define the $n \times d$ design matrix X (whose i^{th} row is $\mathbf{x}_i^\top$), the label n-vector $\mathbf{y} \triangleq [y_1 \quad \ldots \quad y_n]^\top$, and $\mathbf{s} \triangleq [s_1 \quad \ldots \quad s_n]^\top$. For an n-vector $\mathbf{a}$, let $\ln \mathbf{a} \triangleq [\ln a_1 \quad \ldots \quad \ln a_n]^\top$. The cost function can be rewritten in vector form as

$$J(\mathbf{w}) = -\mathbf{y} \cdot \ln \mathbf{s} - (\mathbf{1} - \mathbf{y}) \cdot \ln (\mathbf{1} - \mathbf{s}).$$

Further, recall that for a real symmetric matrix $A \in \mathbb{R}^{d \times d}$, there exist U and Λ such that $A = U \Lambda U^\top$ is the eigendecomposition of A. Here Λ is a diagonal matrix with entries $\{\lambda_1, ..., \lambda_d\}$. An alternative notation is $\Lambda = \text{diag}(\lambda_i)$, where diag() takes as input the list of diagonal entries, and constructs the corresponding diagonal matrix. This notation is widely used in libraries like `numpy`, and is useful for simplifying some of the expressions when written in matrix-vector form. For example, we can write $\mathbf{s} = \text{diag}(s_i)\,\mathbf{1}$.

Hint: See page two for notational conventions used here.

Hint: Recall matrix calculus identities. The elements in **bold** indicate vectors.

$$\nabla_{\mathbf{x}} \alpha \mathbf{y} = \big(\nabla_{\mathbf{x}} \alpha\big)\mathbf{y}^\top + \alpha \nabla_{\mathbf{x}} \mathbf{y} \qquad \nabla_{\mathbf{x}}\big(\mathbf{y} \cdot \mathbf{z}\big) = \big(\nabla_{\mathbf{x}} \mathbf{y}\big)\mathbf{z} + \big(\nabla_{\mathbf{x}} \mathbf{z}\big)\mathbf{y};$$
$$\nabla_{\mathbf{x}} \mathbf{f}(\mathbf{y}) = \big(\nabla_{\mathbf{x}} \mathbf{y}\big)\big(\nabla_{\mathbf{y}} \mathbf{f}(\mathbf{y})\big); \qquad \nabla_{\mathbf{x}} g(\mathbf{y}) = \big(\nabla_{\mathbf{x}} \mathbf{y}\big)\big(\nabla_{\mathbf{y}} g(\mathbf{y})\big);$$

and $\nabla_{\mathbf{x}} C\mathbf{y}(\mathbf{x}) = \big(\nabla_{\mathbf{x}} \mathbf{y}(\mathbf{x})\big)C^\top$, *where C is a constant matrix.*

1 Derive the gradient $\nabla_{\mathbf{w}} J(\mathbf{w})$ of cost $J(\mathbf{w})$ as a matrix-vector expression. Also derive *all intermediate derivatives* in matrix-vector form. Do NOT specify them (**including the intermediates**) in terms of their individual components (e.g. w_i for vector $\mathbf{w}$).

2 Derive the Hessian $\nabla^2_{\mathbf{w}} J(\mathbf{w})$ for the cost function $J(\mathbf{w})$ as a matrix-vector expression.

3 Write the matrix-vector update law for one iteration of Newton's method, substituting the gradient and Hessian of $J(\mathbf{w})$.

4 You are given four examples $\mathbf{x}_1 = [0.2 \quad 3.1]^\top, \mathbf{x}_2 = [1.0 \quad 3.0]^\top, \mathbf{x}_3 = [-0.2 \quad 1.2]^\top, \mathbf{x}_4 = [1.0 \quad 1.1]^\top$ with labels $y_1 = 1, y_2 = 1, y_3 = 0, y_4 = 0$. These points cannot be separated by a line passing through origin. Hence, as described in lecture, append a 1 to each $\mathbf{x}_{i \in [4]}$ and use a weight vector $\mathbf{w} \in \mathbb{R}^3$ whose last component is the bias term (called α in lecture). Begin with initial weight $w^{(0)} = \begin{bmatrix} -1 & 1 & 0 \end{bmatrix}^\top$. For the following, state only the final answer with four digits after the decimal point. You may use a calculator or write a program to solve for these, but do NOT submit any code for this part.

(a) State the value of $\mathbf{s}^{(0)}$ (the initial value of $\mathbf{s}$).

(b) State the value of $\mathbf{w}^{(1)}$ (the value of $\mathbf{w}$ after 1 iteration).

(c) State the value of $\mathbf{s}^{(1)}$ (the value of $\mathbf{s}$ after 1 iteration).

(d) State the value of $\mathbf{w}^{(2)}$ (the value of $\mathbf{w}$ after 2 iterations).

Section 2

1. $x \in \mathbb{R}^d \quad y \in \{0,1\} \quad w \in \mathbb{R}^d$

$s_i = s(x_i \cdot w) \qquad s(y) = \frac{1}{1+e^{-y}}$

$-\nabla J(w) = \nabla_w [-y \cdot \ln s) + \nabla_w ((1-y)\cdot \ln(1-s))$

$\nabla_w(y \cdot z) \qquad \nabla_w(y \cdot z)$

$= (\nabla_w \cancel{-y}^{0}) \ln s + (\nabla_w \ln s) y + \nabla_w \cancel{(1-y)}^{0} \ln(1-s) + \nabla_w (\ln(1-s))(1-y)$

$\nabla_w g(y) \qquad \nabla_x g(y)$

$= \nabla_w(s)(\nabla_s \cancel{\ln s}) \frac{1}{s} y + \nabla_w(1-s) \nabla_{1-s} \cancel{(\ln(1-s))}^{-\frac{1}{1-s}} (1-y)$

$s'(w\cdot x) \qquad s'(w\cdot x)$

$= \nabla(s)\left(\frac{1}{s} y - \frac{1}{1-s}(1-y)\right)$

$= X^T s(1-s)\left(\frac{1}{s} y - \frac{1}{1-s}(1-y)\right)$

$= X^T[s(1-s)\frac{1}{s} y - s(1-s)\frac{1}{1-s}(1-y)]$

$= X^T[s(\frac{1}{s} - \vec{1}) y - s(1-y)]$

$= X^T[(\vec{1} - s) y - s + sy]$

$= X^T[y - sy - s + sy]$

$-\nabla J(w) = X^T(y - s)$

2. $\nabla J(w) = -X^T(y-s)$

$\nabla^2 J(w) = -\cancel{X^T y}^{0} + X^T s$

$= (\nabla_x s) X$

$= X^T \, diag(s(1-s)) \, X$

3. $w = v + \left((X^T \Omega X)^{-1} \cdot (X^T(y-s))\right)$

4. a) $s^{(0)} \doteq \begin{bmatrix} 0.948 \\ 0.881 \\ 0.602 \\ 0.525 \end{bmatrix}$ b) $w^{(1)} \doteq \begin{bmatrix} 1.325 \\ 3.050 \\ -6.829 \end{bmatrix}$

c) $s^{(1)} = \begin{bmatrix} 0.947 \\ 0.975 \\ 0.031 \\ 0.104 \end{bmatrix}$ d) $w^{(2)} \doteq \begin{bmatrix} 1.366 \\ 4.158 \\ -9.200 \end{bmatrix}$

3 Wine Classification with Logistic Regression

The wine dataset `data.mat` consists of 6,000 sample points, each having 12 features. The description of these features is provided in `data.mat`. The dataset includes a training set of 5,000 sample points and a test set of 1,000 sample points. Your classifier needs to predict whether a wine is white (class label 0) or red (class label 1).

Begin by normalizing the data with each feature's mean and standard deviation. You should use training data statistics to normalize both training and validation/test data. Then add a fictitious dimension. Whenever required, it is recommended that you tune hyperparameter values with cross-validation.

Please set a random seed whenever needed and **report it**.

Use of automatic logistic regression libraries/packages is prohibited for this question. If you are coding in python, it is better to use `scipy.special.expit` for evaluating logistic functions as its code is numerically stable, and doesn't produce `NaN` or `MathOverflow` exceptions.

1 *Batch Gradient Descent Update*. State the batch gradient descent update law for logistic regression **with ℓ_2 regularization**. As this is a "batch" algorithm, each iteration should use *every training example*. You don't have to show your derivation. You may reuse results from your solution to question 2.1.

2 *Batch Gradient Descent Code*. Implement your batch gradient descent algorithm for logistic regression and include your code here. Choose reasonable values for the regularization parameter and step size (learning rate), specify your chosen values in the write-up, and train your model from question 3.1. Shuffle and split your data into training/validation sets and mention the random seed used in the write-up. Plot the value of the cost function versus the number of iterations spent in training.

3 *Stochastic Gradient Descent (SGD) Update*. State the SGD update law for logistic regression with ℓ_2 regularization. Since this is not a "batch" algorithm anymore, each iteration uses *just one* training example. You don't have to show your derivation.

4 *Stochastic Gradient Descent Code*. Implement your stochastic gradient descent algorithm for logistic regression and include your code here. Choose a suitable value for the step size (learning rate), specify your chosen value in the write-up, and run your SGD algorithm from question 3.3. Shuffle and split your data into training/validation sets and mention the random seed used in the write-up. Plot the value of the cost function versus the number of iterations spent in training.

   Compare your plot here with that of question 3.2. Which method converges more quickly? Briefly describe what you observe.

5 Instead of using a constant step size (learning rate) in SGD, you could use a step size that slowly shrinks from iteration to iteration. Run your SGD algorithm from question 3.3 with a step size $\epsilon_t = \delta/t$ where t is the iteration number and δ is a hyperparameter you select

empirically. Mention the value of δ chosen. Plot the value of cost function versus the number of iterations spent in training.

How does this compare to the convergence of your previous SGD code?

6 *Kaggle*. Train your *best* classifier on the entire training set and submit your prediction on the test sample points to Kaggle. As always for Kaggle competitions, you are welcome to add or remove features, tweak the algorithm, and do pretty much anything you want to improve your Kaggle leaderboard performance **except** that you may not replace or augment logistic regression with a wholly different learning algorithm. Your code should output the predicted labels in a CSV file.

Report your Kaggle username and your best score, and briefly describe what your best classifier does to achieve that score.

Section 3:

1. Regular cost fn

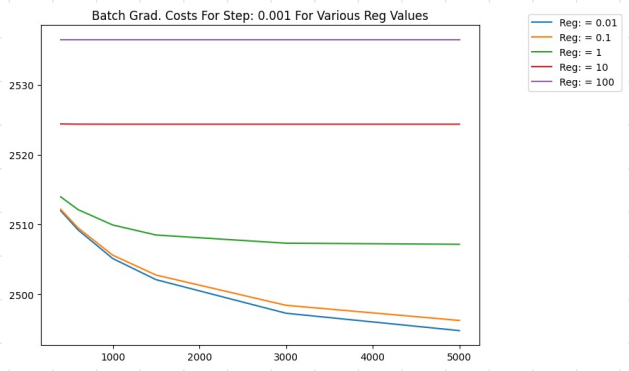
$$J(w) = -\sum_{i=1}^{n} \log(L(s(x_i \cdot w), y_i))$$

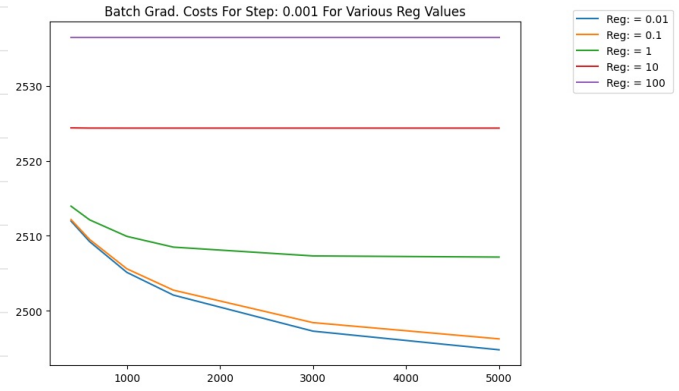
regularized: $J(w) =$ $\quad + \lambda \sum_{j=1}^{n} w_j^2$

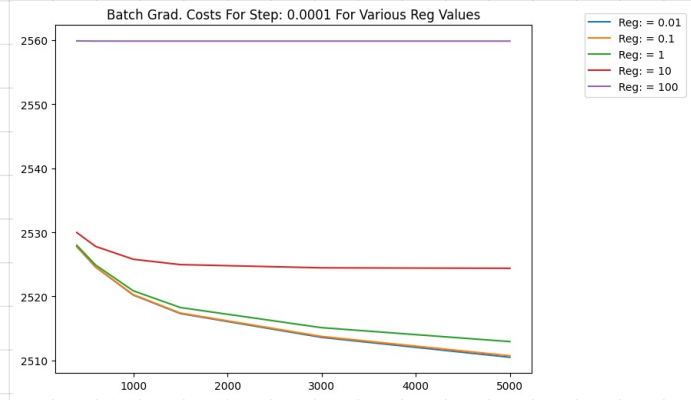
Batch Gradient: $\nabla_w J(w) = -\sum_{j=1}^{m} (y_i - s_i) x_i + \lambda w_i$

$$= -X^T(y - s(Xw)) + \lambda \vec{w}$$

2.







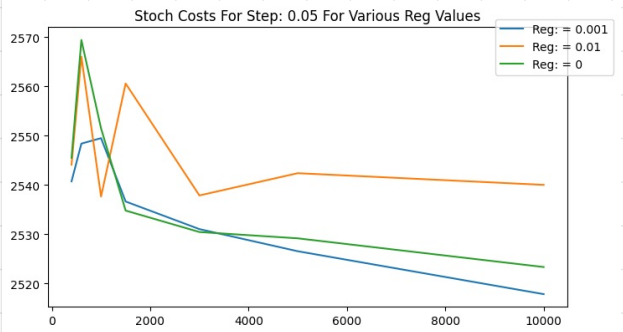
Step 0.001, Reg 0.01 is best

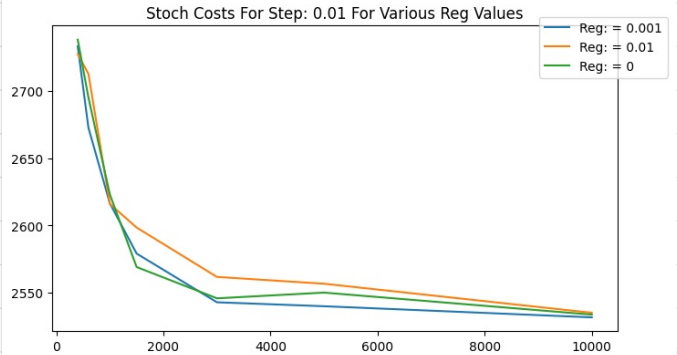
Validation: 0.951

3. ~~Batch~~ Stocastic Gradient : $\nabla_w J(w) = -(y_i - s_i)x_i + \lambda w_i$

where y_i, s_i, x_i, w_i are a randomly selected training point, weight, and sigmoid.

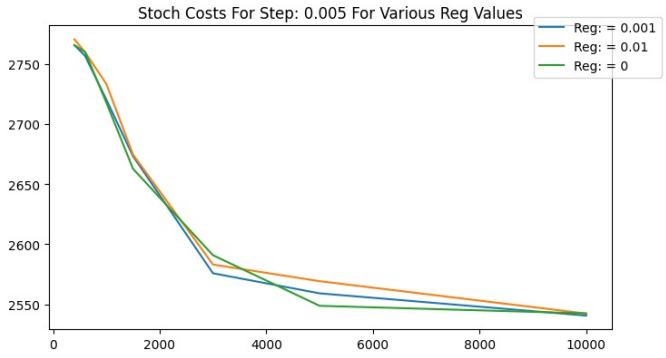
4. seed = 0, Seeded at the beginning of code.

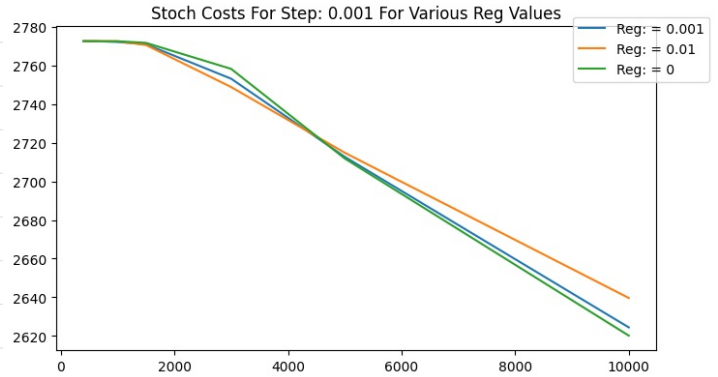




Step 0.01, Reg 0.001 is best

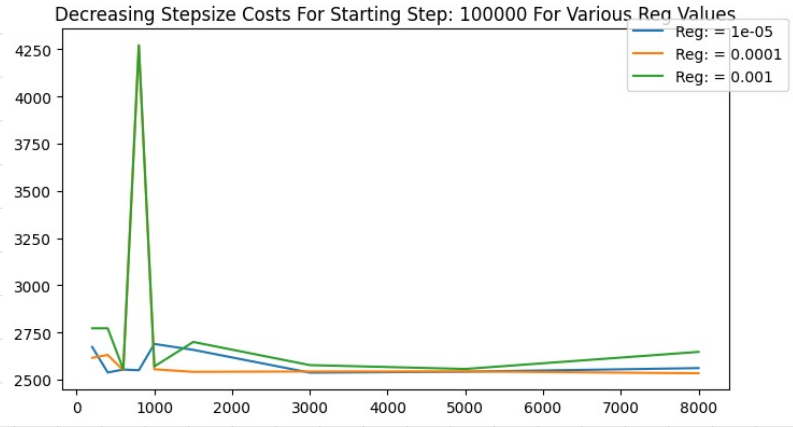
Validation: 0.924

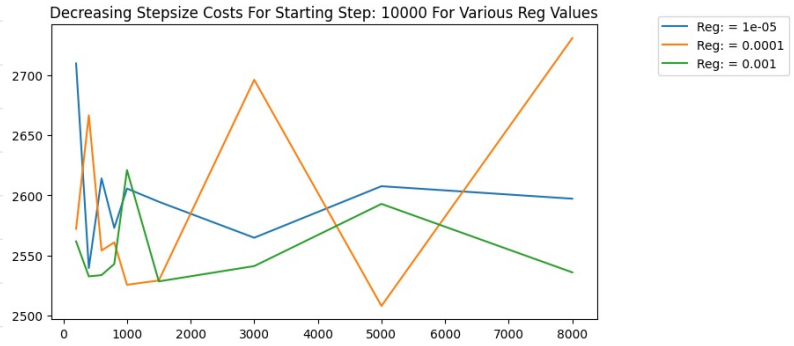


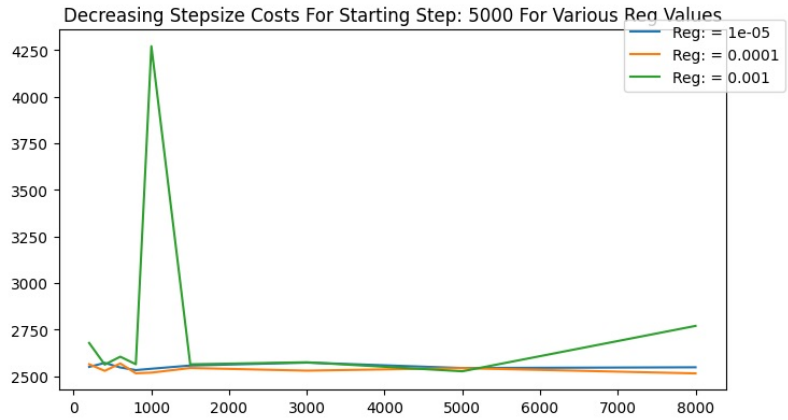


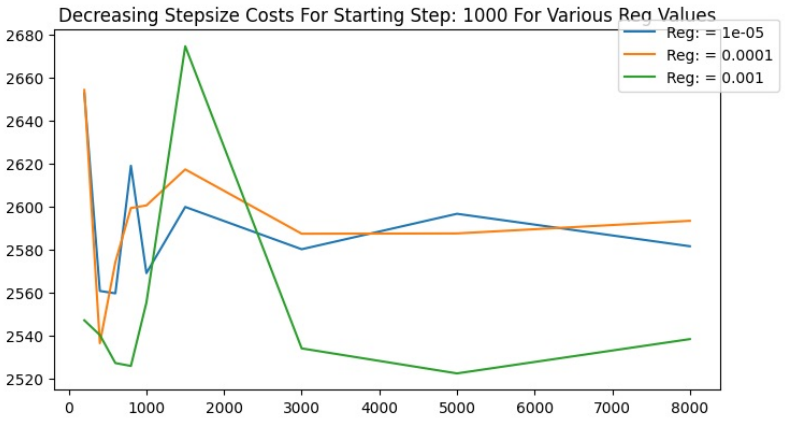
The plots here are a lot less smooth. They seem to converge with fewer iterations, and run much faster. However, the cost function doesn't go down as much

5.









Step: 100000 , Reg: 0.00001 is best.

Validation: 0.940

Decreasing step size is good for quickly decreasing cost. Smaller step later on makes convergence faster.

6. Kaggle Username: Frank Jin (fj0228)

Best score: 0.95

I found that using regular batch gradient descent gave me the best result.

4 A Bayesian Interpretation of Lasso

Suppose you are aware that the labels $y_{i\in[n]}$ corresponding to sample points $\mathbf{x}_{i\in[n]} \in \mathbb{R}^d$ follow the density law

$$f(y_i|\mathbf{x}_i, \mathbf{w}) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(y_i - \mathbf{w}\cdot\mathbf{x}_i)^2/(2\sigma^2)}$$

where $\sigma > 0$ is a known constant and $\mathbf{w} \in \mathbb{R}^d$ is a random parameter. Suppose further that experts have told you that

- each component of $\mathbf{w}$ is independent of the others, and
- each component of $\mathbf{w}$ has the Laplace distribution with location 0 and scale being a known constant b. That is, each component $\mathbf{w}_i$ obeys the density law $f(\mathbf{w}_i) = e^{-|\mathbf{w}_i|/b}/(2b)$.

Assume the outputs $y_{i\in[n]}$ are independent from each other.

Your goal is to find the choice of parameter $\mathbf{w}$ that is *most likely* given the input-output examples $(\mathbf{x}_i, y_i)_{i\in[n]}$. This method of estimating parameters is called *maximum a posteriori* (MAP); Latin for *"maximum [odds] from what follows."*

1. Derive the *posterior* probability density law $f(\mathbf{w}|(\mathbf{x}_i, y_i)_{i\in[n]})$ for $\mathbf{w}$ *up to a proportionality constant* by applying Bayes' Theorem and substituting for the densities $f(y_i|\mathbf{x}_i, \mathbf{w})$ and $f(\mathbf{w})$. Don't try to derive an exact expression for $f(\mathbf{w}|(\mathbf{x}_i, y_i)_{i\in[n]})$, as the denominator is very involved and irrelevant to maximum likelihood estimation.

2. Define the log-likelihood for MAP as $\ell(\mathbf{w}) \triangleq \ln f(\mathbf{w}|\mathbf{x}_{i\in[n]}, y_{i\in[n]})$. Show that maximizing the MAP log-likelihood over all choices of $\mathbf{w}$ is the same as minimizing $\sum_{i=1}^n (y_i - \mathbf{w}\cdot\mathbf{x}_i)^2 + \lambda\|\mathbf{w}\|_1$ where $\|\mathbf{w}\|_1 = \sum_{j=1}^d |w_j|$ and λ is a constant. Also give a formula for λ as a function of the distribution parameters.

Section 4

1.

$$f(w \mid (x_i, y_i)_{i \in [n]}) \propto f((x_i, y_i)_{i \in [n]} \mid w) \cdot f(w)$$

($f(w)$)

$$= \prod_{i=1}^{n} f(y_i \mid x_i, w) \cdot \prod_{j=1}^{d} \frac{1}{2b} e^{-\frac{|w_j|}{b}}$$

($f(y_i \mid x_i, w)$: defined in question)

2. $l(w) = \ln(f(w \mid (x_i, y_i)_{i=1}^{n}))$

$$\propto \sum_{i=1}^{n} \left(-\frac{(y_i - w \cdot x_i)^2}{2\sigma^2} \right) - \sum_{j=1}^{d} \frac{|w_j|}{b}$$

$$= -\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - w \cdot x_i)^2 - \sum_{j=1}^{d} \frac{|w_j|}{b}$$

max x = min $-x$

$$\min \left(\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - w \cdot x_i)^2 - \sum_{j=1}^{d} \frac{|w_j|}{b} \right)$$

($\sum_{j=1}^{d} \frac{|w_j|}{b} = \lambda \|w\|_1$)

$$= \min \left(\sum_{i=1}^{n} (y_i - w \cdot x_i)^2 - \lambda \|w\|_1 \right)$$

5 ℓ_1-regularization, ℓ_2-regularization, and Sparsity

You are given a design matrix X (whose i^{th} row is sample point $\mathbf{x}_i^\top$) and an n-vector of labels $\mathbf{y} \triangleq [y_1 \quad \dots \quad y_n]^\top$. For simplicity, assume X is whitened, so $X^\top X = nI$. Do not add a fictitious dimension/bias term; for input $\mathbf{0}$, the output is always 0. Let $\mathbf{x}_{*i}$ denote the i^{th} column of X.

1. The ℓ_p-norm for $w \in \mathbb{R}^d$ is defined as $\|w\|_p = (\sum_{i=1}^d |w_i|^p)^{1/p}$, where $p > 0$. Plot the isocontours with $w \in \mathbb{R}^2$, for the following norms.
   (a) $\ell_{0.5}$ (b) ℓ_1 (c) ℓ_2
   Use of automatic libraries/packages for computing norms is prohibited for the question.

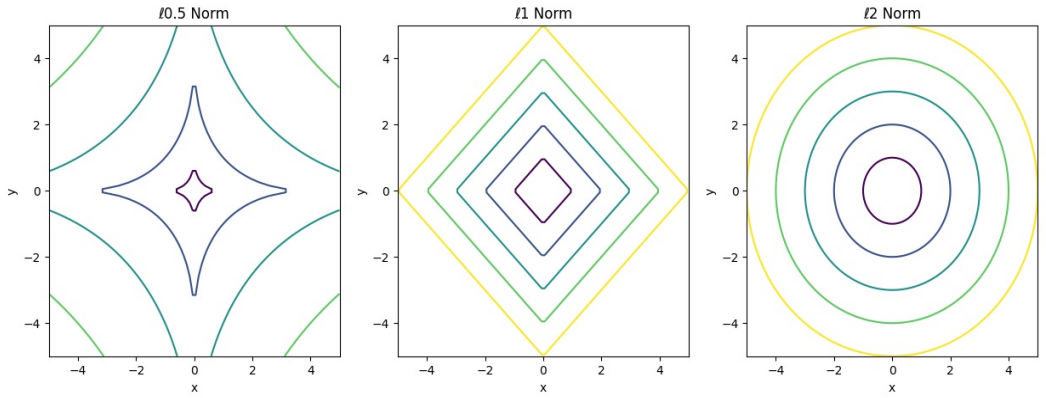
2. Show that the cost function for ℓ_1-regularized least squares, $J_1(\mathbf{w}) \triangleq \|X\mathbf{w} - \mathbf{y}\|^2 + \lambda\|\mathbf{w}\|_1$ (where $\lambda > 0$), can be rewritten as $J_1(\mathbf{w}) = \|\mathbf{y}\|^2 + \sum_{i=1}^d f(\mathbf{x}_{*i}, \mathbf{w}_i)$ where $f(\cdot,\cdot)$ is a suitable function whose first argument is a vector and second argument is a scalar.

3. Using your solution to part 2, derive necessary and sufficient conditions for the i^{th} component of the optimizer $\mathbf{w}^*$ of $J_1(\cdot)$ to satisfy each of these three properties: $w_i^* > 0, w_i^* = 0$, and $w_i^* < 0$.

4. For the optimizer $\mathbf{w}^\#$ of the ℓ_2-regularized least squares cost function $J_2(\mathbf{w}) \triangleq \|X\mathbf{w} - \mathbf{y}\|^2 + \lambda\|\mathbf{w}\|^2$ (where $\lambda > 0$), derive a necessary and sufficient condition for $\mathbf{w}_i^\# = 0$, where $\mathbf{w}_i^\#$ is the ith component of $\mathbf{w}^\#$.

5. A vector is called *sparse* if most of its components are 0. From your solution to part 3 and 4, which of $\mathbf{w}^*$ and $\mathbf{w}^\#$ is more likely to be sparse? Why?

1.



a) b) c)

2. $J_1(w) = (Xw - y)^T(Xw - y) + \lambda\|w\|_1$

$= w^T X^T X w - w^T X^T y - y^T X w + y^T y + \lambda\|w\|_1$

$= n\|w\|^2 - 2w^T X^T y + \|y\|^2 + \lambda\|w\|_1$

$= \|y\|^2 + \sum_{i=1}^{d} \underbrace{(n|w_i|^2 - 2x_i^* y_i + \lambda|w_i|)}_{f(x_i^*, w_i)}$

3. $\frac{\partial J_1(w)}{\partial w_i} = 2nw_i + \lambda$

For $w_i^* > 0$

$\frac{\partial J_1(w)}{\partial w_i} > 0$

< 0

$\frac{\partial J_1(w)}{\partial w_i} < 0$

$= 0$

$\frac{\partial J_1(w)}{\partial w_i} = 0$

4. $\frac{\partial J_2(w)}{\partial w_i} = 2X^T Xw - 2X^T y + 2\lambda w = 0$

$w_i = \frac{(X^T y)_i}{n+\lambda} = 0$

$(X^T y)_i = 0$

So $(X^T y)_i = 0$ for w_i^* to be 0.

5. l_1 is more sparse as it penalizes non-zero components of w. Whereas l_2 distributes weights more evenly.

Submission Checklist

Please ensure you have completed the following before your final submission.

At the beginning of your writeup...

1. Have you copied and hand-signed the honor code specified in Question 1?
2. Have you listed all students (Names and ID numbers) that you collaborated with?

In your writeup for Question 3...

1. Have you included your **Kaggle Score** and **Kaggle Username**?

At the end of the writeup...

1. Have you provided a code appendix including all code you wrote in solving the homework?

Executable Code Submission

1. Have you created an archive containing all ".py" files that you wrote or modified to generate your homework solutions?
2. Have you removed all data and extraneous files from the archive?
3. Have you included a `README` file in your archive containing any special instructions to reproduce your results?

Submissions

1. Have you submitted your written solutions to the Gradescope assignment titled **HW4 Write-Up** and selected pages appropriately?
2. Have you submitted your executable code archive to the Gradescope assignment titled **HW4 Code**?
3. Have you submitted your test set predictions for **Wine** dataset to the appropriate Kaggle challenge?

Congratulations! You have completed Homework 4.

Q3

March 8, 2024

1 Appendix

1.1 Question 3

[ ]:
```
#Imports
import numpy as np
import scipy.io as io
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from scipy.special import expit
from sklearn.model_selection import GridSearchCV
```

[ ]:
```
SEED = 2

#Seeding RNG with seed = 0
rng = np.random.Generator(np.random.PCG64(seed=SEED))
```

1.1.1 Part 2

[ ]:
```
data = io.loadmat('data.mat')
#print(data['description'])
samples = data['X']
labels = data['y']
test = data['X_test']

print("samples: ", samples.shape)
print("labels: ", labels.shape)
```

```
samples:  (5000, 12)
labels:  (5000, 1)
```

[ ]:
```
#Split
xTrainRaw, xTestRaw, yTrain, yTest = train_test_split(samples, labels,␣
 ↪test_size=0.20, random_state=SEED, shuffle=True)

print("xTrain: ", xTrainRaw.shape)
print("xTest: ", xTestRaw.shape)
print("yTrain: ", yTrain.shape)
```

```
print("yTest: ", yTest.shape)
```

```
xTrain:  (4000, 12)
xTest:  (1000, 12)
yTrain:  (4000, 1)
yTest:  (1000, 1)
```

[ ]:
```
#Data editing
#Normalize data
xTrain = (xTrainRaw - xTrainRaw.mean()) / xTrainRaw.std()
xTest = (xTestRaw - xTrainRaw.mean()) / xTrainRaw.std()
test = (test - xTrainRaw.mean()) / xTrainRaw.std()

#Adding ficticious dim
ones = np.ones((xTrain.shape[0], 1))
xTrain = np.hstack((xTrain, ones))
print("xTrain with fict dim: ", xTrain.shape)

ones = np.ones((xTest.shape[0], 1))
xTest = np.hstack((xTest, ones))
print("xTest with fict dim: ", xTrain.shape)

ones = np.ones((test.shape[0], 1))
test = np.hstack((test, ones))
print("test data with fict dim: ", test.shape)
```

```
xTrain with fict dim:  (4000, 13)
xTest with fict dim:  (4000, 13)
test data with fict dim:  (1000, 13)
```

[ ]:
```
def validate(w, xTest, yTest):
    validationY = np.round(expit(xTest.dot(w)))
    validationY = np.reshape(validationY, (validationY.shape[0]))
    print(yTest.shape)
    yTest = np.reshape(yTest, (yTest.shape[0]))
    # print(validationY.reshape(1, validationY.shape[0]))
    # print(yTest.reshape(1, validationY.shape[0]))
    correct = 0
    total = len(validationY)
    for i in range(0, len(yTest)):
        if validationY[i] == yTest[i]:
            correct += 1
    return correct/total
```

[ ]:
```
#Batch Gradient
def batchGrad(x, y, w, reg):
    result = y - expit(x.dot(w)) #logisticFn(x, w)
    result = -x.T.dot(result) + reg * w
```

```
    return result

#Take step
def update(w, step, grad):
    w = w - step * grad
    return w

def cost(x, y, w):
    result = y * np.log(expit(np.clip(x.dot(w), a_min = 0.0001,  a_max=1)))
    result = result + (1 - y) * np.log(1 - expit(np.clip(x.dot(w), a_min = 0.
 ↪0001, a_max=1 )))
    result = np.sum(result)
    return -result
```

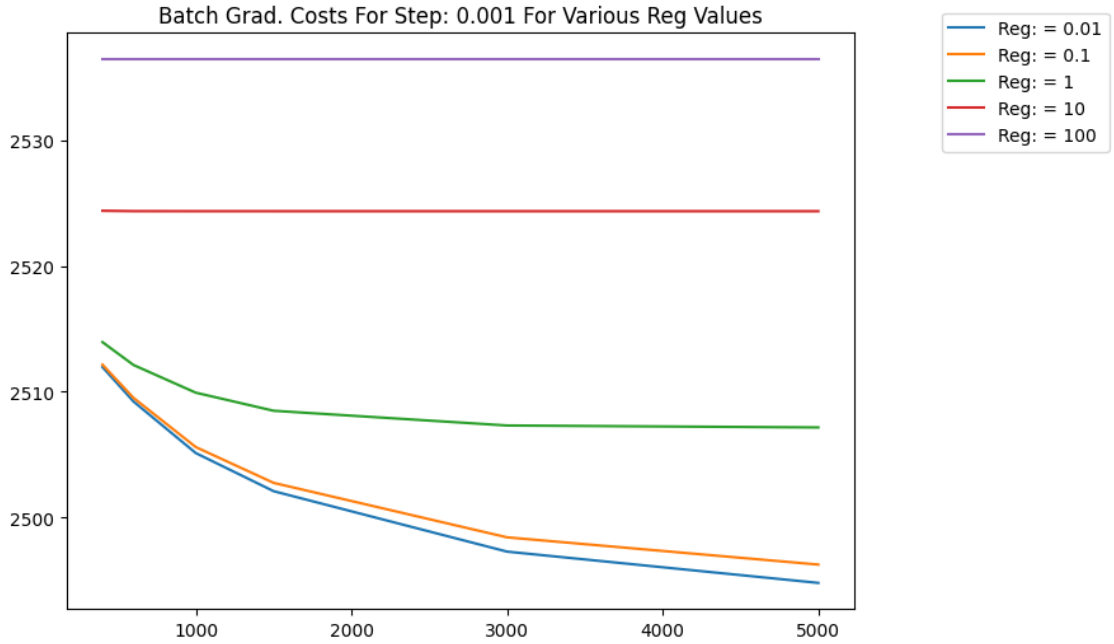
[ ]:
```
def train(tolerance, max_iter, gradFn, trainSize, step, reg):
    #Start guess at 0
    w = np.zeros((1, xTrain.shape[1])).T

    #Find first gradient
    iter = 0

    while iter < max_iter:
        grad = gradFn(xTrain[:trainSize], yTrain[:trainSize], w, reg)
        w = update(w, step, grad)
        iter += 1
        if np.all(abs(grad)) < tolerance:
            print("Met Tolarence")
            break
    return w
```

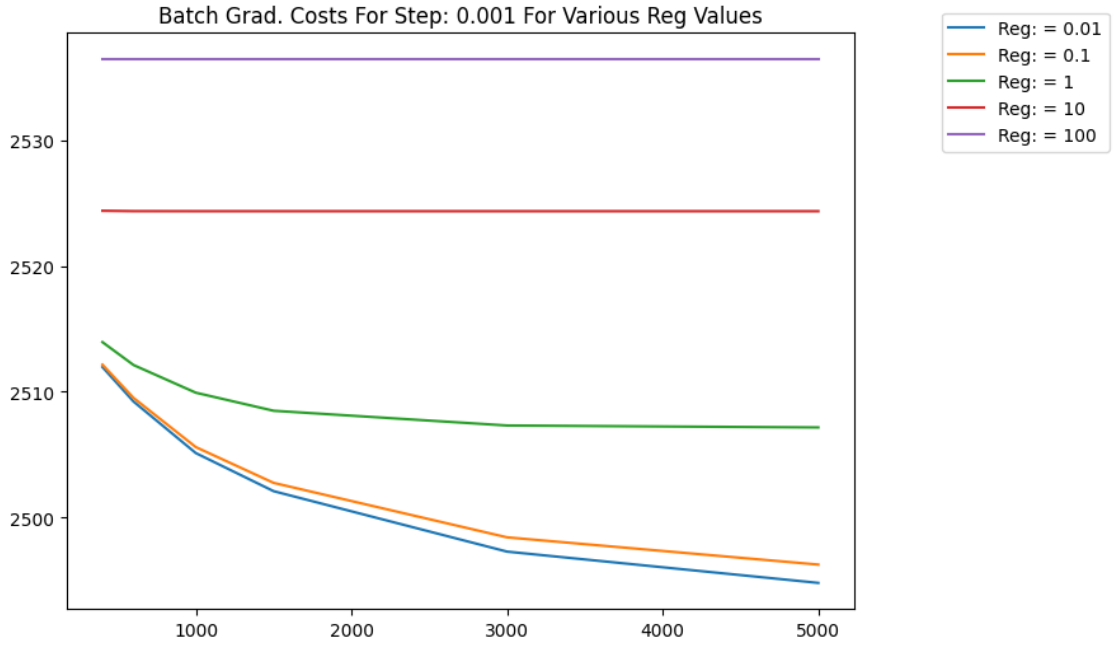
[ ]:
```
iter = [400, 600, 1000, 1500, 3000, 5000]
stepSize = [0.001, 0.001, 0.0001]
reg = [0.01, 0.1, 1, 10, 100]
batchCostsForStep = []
# param_grid = [
#   {'reg': [0.001, 0.01, 0.1, 1, 10], 'stepSize' : [0.001, 0.001, 0.0001]},
#   {'iter': [100, 200, 400, 600, 1000, 1500, 3000, 5000]},
#  ]
for step in stepSize:
    costDict = {}
    for r in reg:
        currCost = []
        for i in iter:
            w = train(0.1, i, batchGrad, 4000, step, r)
            c = cost(xTrain[:4000], yTrain[:4000], w)
            currCost.append(c)
        costDict[f'Reg: = {r}'] = currCost
```

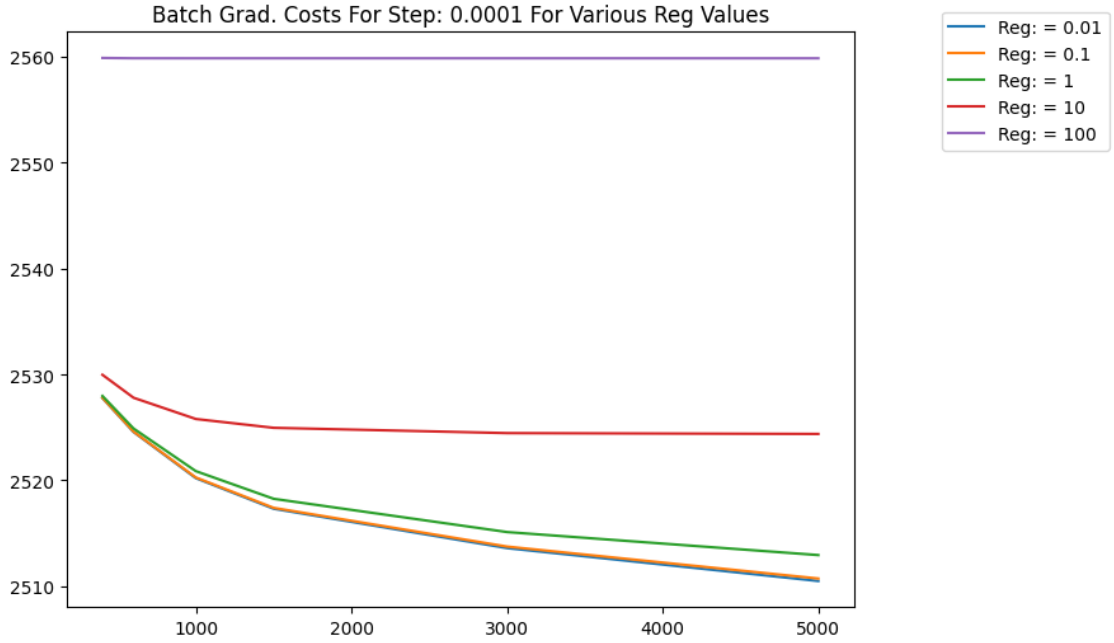
```
    batchCostsForStep.append(costDict)
```

```
Met Tolarence
Met Tolarence
Met Tolarence
Met Tolarence
```

[ ]:
```
for i in range(0, len(stepSize)):
    plt.figure(figsize=(8, 20))
    for key in batchCostsForStep[i].keys():
        plt.subplot(3, 1, i+1)
        plt.plot(iter, batchCostsForStep[i][key], label = key)
        plt.title(f"Batch Grad. Costs For Step: {stepSize[i]} For Various Reg␣
 ↪Values")
        plt.legend(bbox_to_anchor=(1.1, 1.05))
plt.show()
```







[ ]:
```
w = train(0.1, 50000, batchGrad, 4000, 0.001, 0.01)
validate(w, xTest, yTest)
```

```
(1000, 1)
```

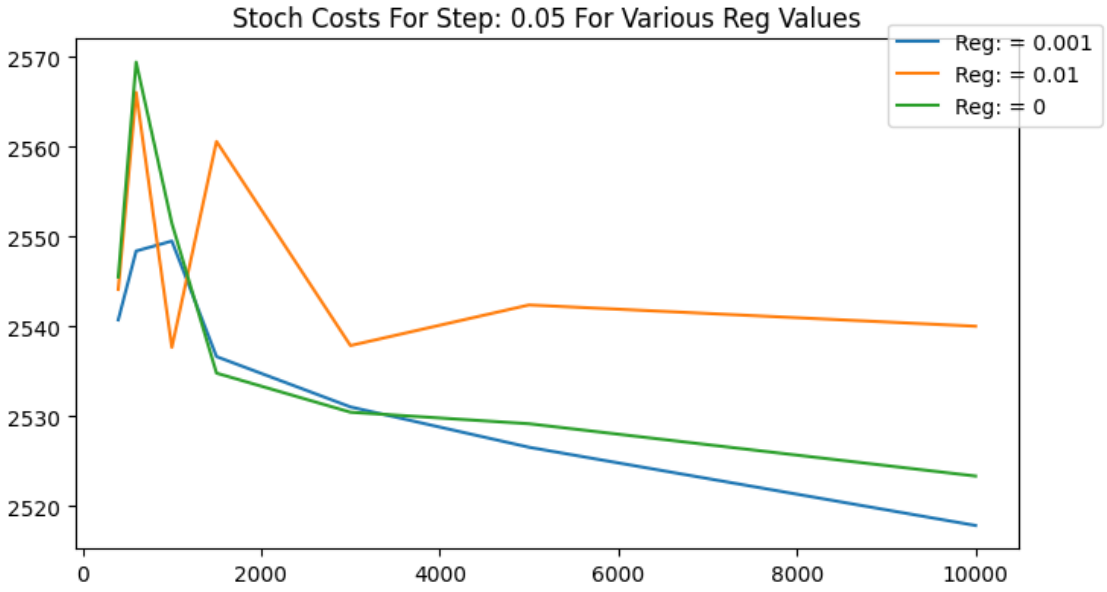
[ ]:
```
0.95
```

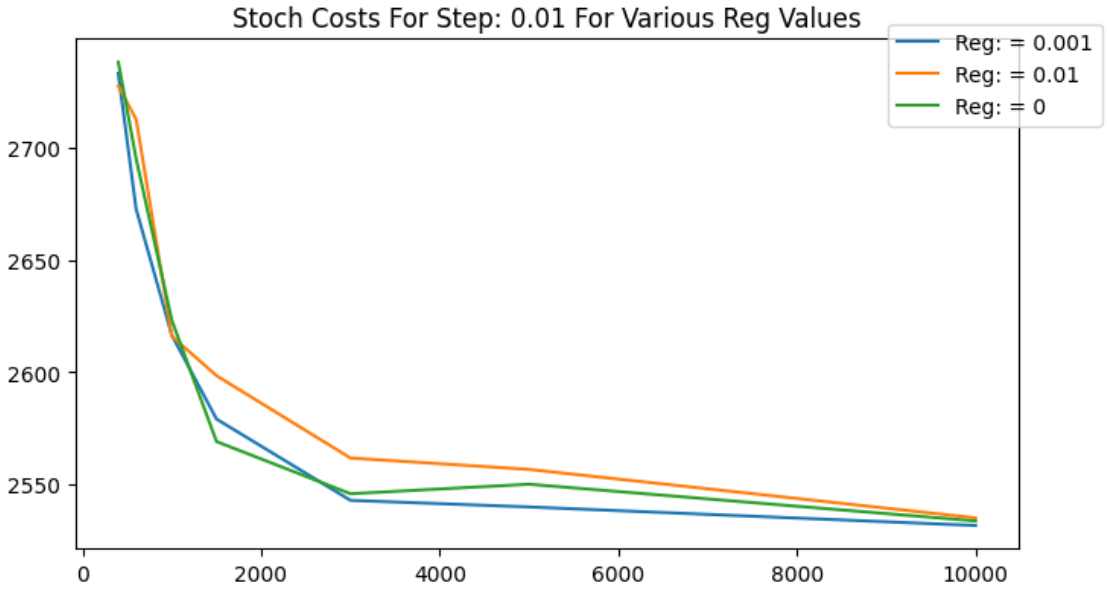
1.1.2 Section 4

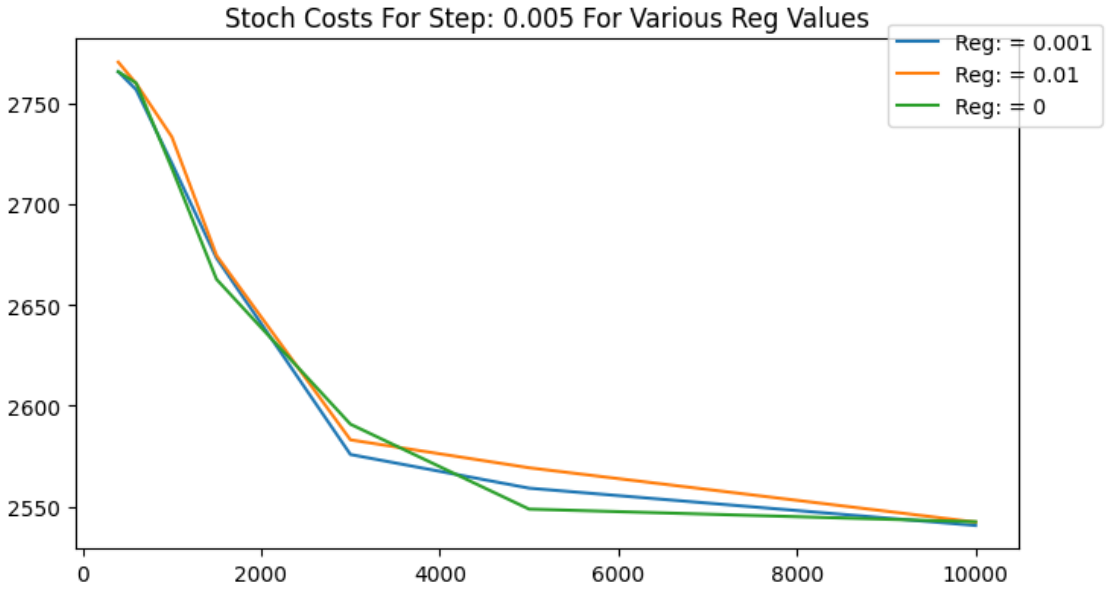
[ ]:
```
#Stochastic Gradient
def stoGrad(x, y, w, reg):
    randIdx = rng.integers(0, x.shape[0])

    xi = x[randIdx]
    yi = y[randIdx]

    result = yi - expit(xi.dot(w)) #logisticFn(x, w)
    result = np.reshape((result * xi), (13,1))
    # print((reg*w).shape)
    # print(result.shape)
    result = -result + reg * w
    return result
```

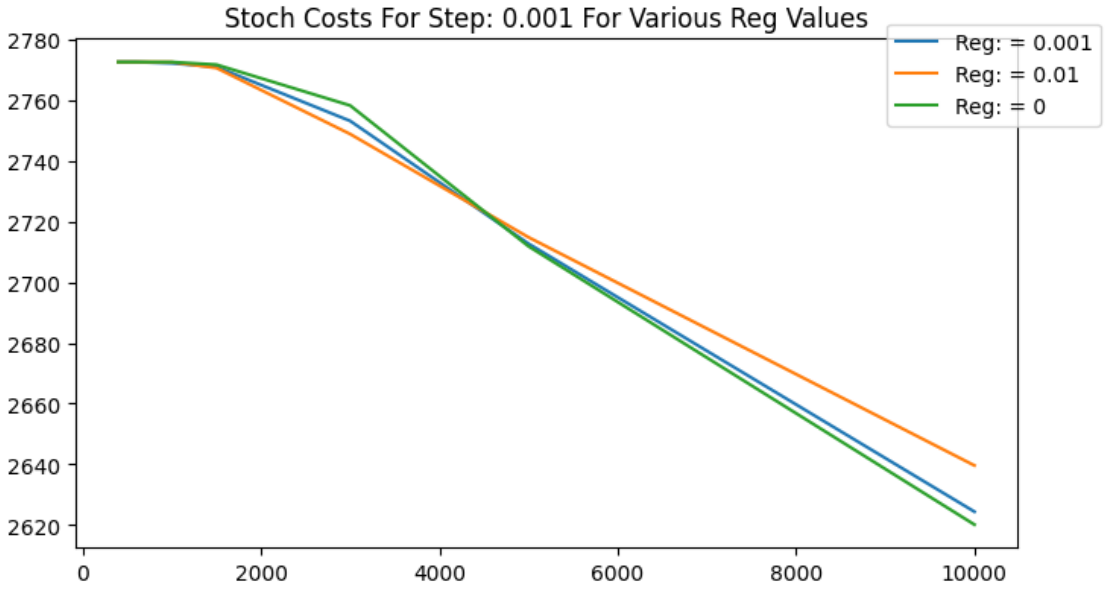
[ ]:
```
#w = train(0.1, 20, stoGrad, 4000, 0.0001, 0)

iter = [400, 600, 1000, 1500, 3000, 5000, 10000, 50000]
stepSize = [0.05, 0.01, 0.005, 0.001]
reg = [0.001, 0.01, 0]
stoCostsForStep = []
for step in stepSize:
    costDict = {}
    for r in reg:
        currCost = []
        for i in iter:
            w = train(0.01, i, stoGrad, 4000, step, r)
            c = cost(xTrain[:4000], yTrain[:4000], w)
            currCost.append(c)
        costDict[f'Reg: = {r}'] = currCost
    stoCostsForStep.append(costDict)
```

[ ]:
```
for i in range(0, len(stepSize)):
    plt.figure(figsize=(8, 20))
    for key in stoCostsForStep[i].keys():
        plt.subplot(len(stepSize), 1, i+1)
        plt.plot(iter, stoCostsForStep[i][key], label = key)
        plt.title(f"Stoch Costs For Step: {stepSize[i]} For Various Reg Values")
        plt.legend(bbox_to_anchor=(1.1, 1.05))
plt.show()
```


Stoch Costs For Step: 0.05 For Various Reg Values
Reg: = 0.001
Reg: = 0.01
Reg: = 0
2570
2560
2550
2540
2530
2520
0
2000
4000
6000
8000
10000


Stoch Costs For Step: 0.01 For Various Reg Values
Reg: = 0.001
Reg: = 0.01
Reg: = 0
2700
2650
2600
2550
0
2000
4000
6000
8000
10000





[ ]:
```
w1 = train(0.1, 100000, stoGrad, 4000, 0.01, 0.001)
validate(w1, xTest, yTest)
```

```
(1000, 1)
```

[ ]: 0.924

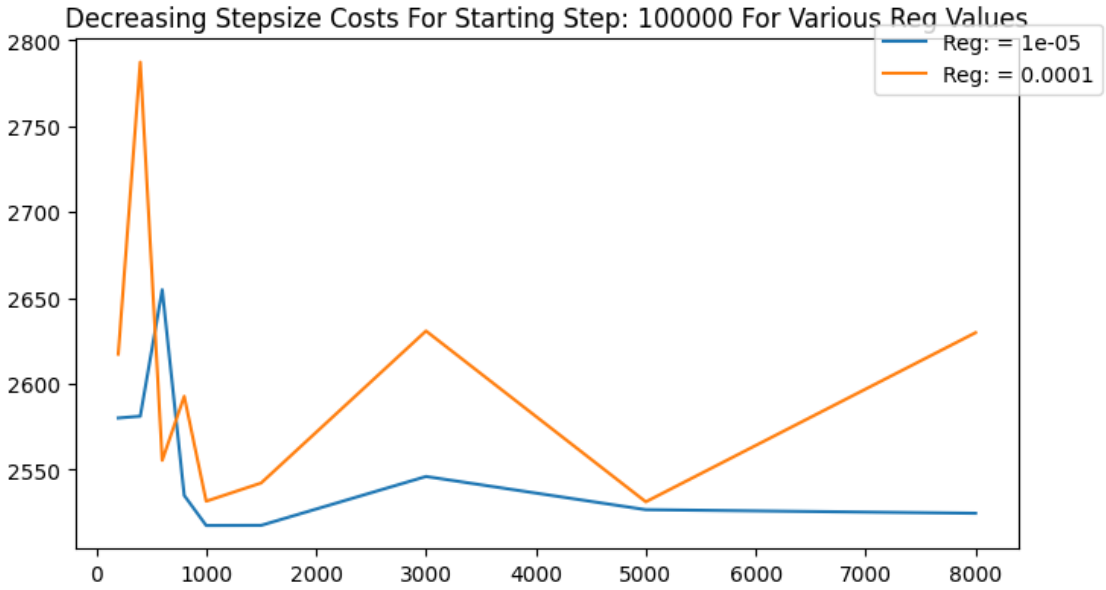
1.1.3 Section 5

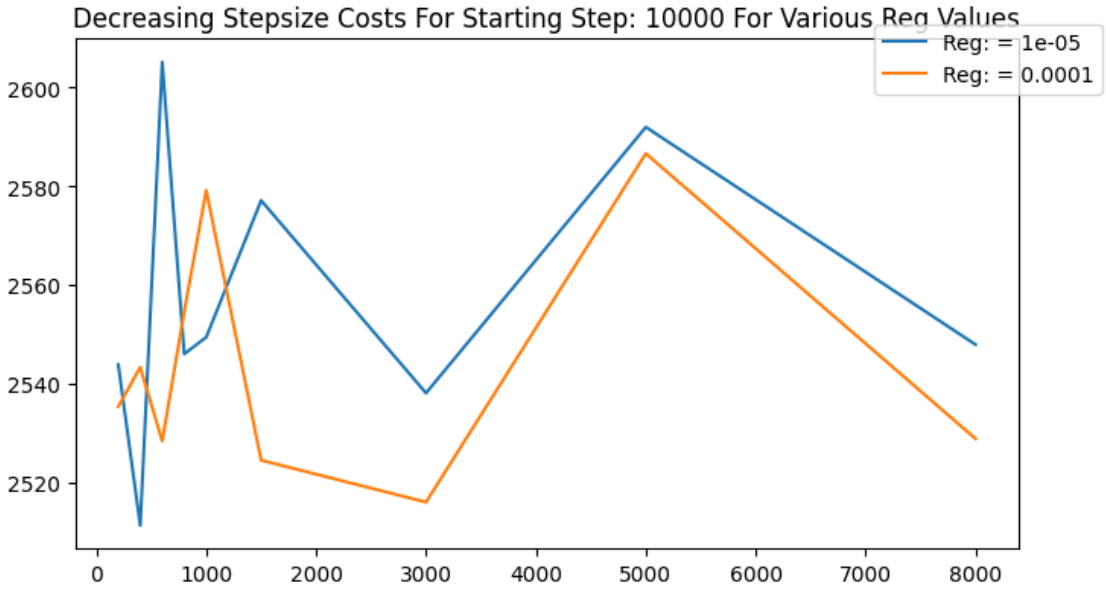
[ ]:
```
#Take step
def decreasingUpdate(w, startingStep, iter, grad):
    #w = w - max(startingStep/iter, 0.0001) * grad
    w = w - startingStep/iter * grad
    return w
```

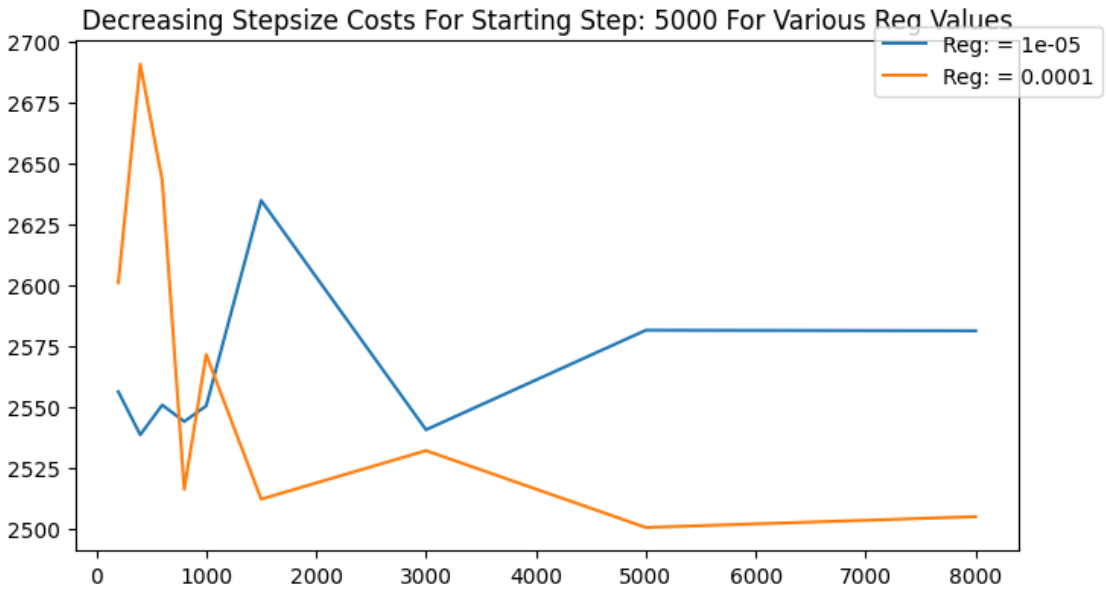
[ ]:
```
def decreasingTrain(tolerance, max_iter, gradFn, trainSize, startingStep, reg):
    #Start guess at 0
    w = np.zeros((1, xTrain.shape[1])).T

    #Find first gradient
    iter = 1

    while iter < max_iter:
        grad = gradFn(xTrain[:trainSize], yTrain[:trainSize], w, reg)
        w = decreasingUpdate(w, startingStep, iter, grad)
        iter += 1
        if np.all(abs(grad)) <= tolerance:
            print("Met Tolarence")
            break
    return w
```

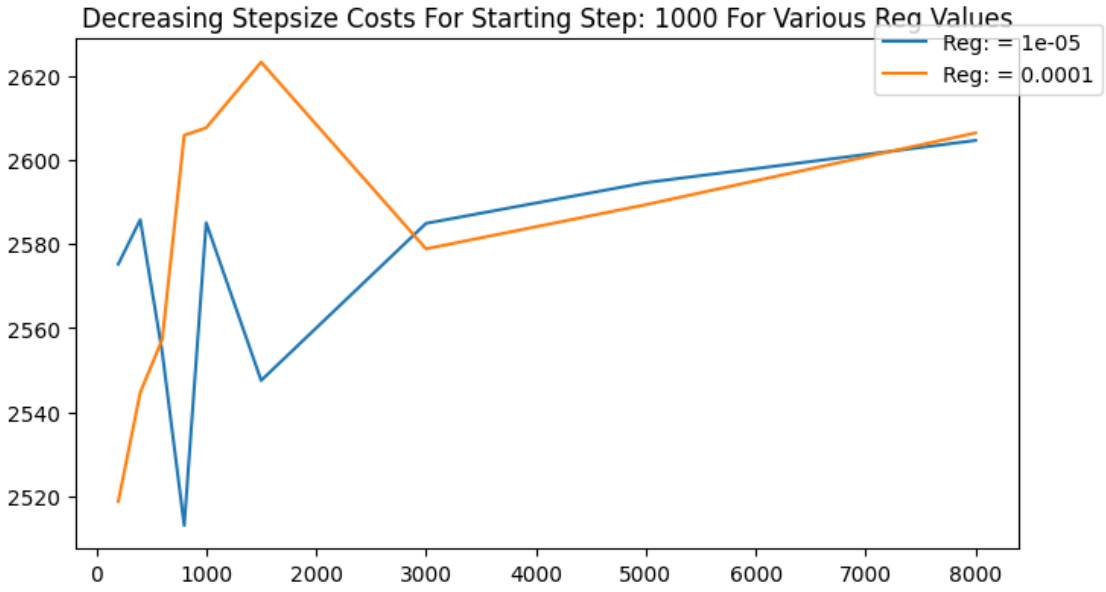
[ ]:
```
iter = [200, 400, 600, 800, 1000, 1500, 3000, 5000, 8000]
startingStep = [100000, 10000, 5000, 1000]
reg = [0.00001, 0.0001]
decreasingCostsForStep = []
for step in startingStep:
    costDict = {}
    for r in reg:
        currCost = []
        for i in iter:
            w = decreasingTrain(0, i, stoGrad, 4000, step, r)
            c = cost(xTrain[:4000], yTrain[:4000], w)
            currCost.append(c)
        costDict[f'Reg: = {r}'] = currCost
    decreasingCostsForStep.append(costDict)
```

[ ]:
```
for i in range(0, len(startingStep)):
    plt.figure(figsize=(8, 20))
    for key in decreasingCostsForStep[i].keys():
        plt.subplot(len(startingStep), 1, i+1)
        plt.plot(iter, decreasingCostsForStep[i][key], label = key)
        plt.title(f"Decreasing Stepsize Costs For Starting Step:␣
 ↪{startingStep[i]} For Various Reg Values")
        plt.legend(bbox_to_anchor=(1.1, 1.05))
plt.show()
```


Decreasing Stepsize Costs For Starting Step: 100000 For Various Reg Values
Reg: = 1e-05
Reg: = 0.0001
2800
2750
2700
2650
2600
2550
0
1000
2000
3000
4000
5000
6000
7000
8000


Decreasing Stepsize Costs For Starting Step: 10000 For Various Reg Values
Reg: = 1e-05
Reg: = 0.0001
2600
2580
2560
2540
2520
0
1000
2000
3000
4000
5000
6000
7000
8000





[ ]:
```
w = decreasingTrain(0.00001, 100000, stoGrad, 4000, 100000, 0.00001)
```

[ ]:
```
validate(w, xTest, yTest)
```

```
(1000, 1)
```

[ ]: 0.94

[ ]:
```
final = train(0.1, 50000, batchGrad, 4000, 0.001, 0.01)
validate(final, xTest, yTest)
```

```
(1000, 1)
```

[ ]:
```
0.95
```

[ ]:
```
result = np.round(expit(test.dot(final)))
print(result.shape)
result = np.reshape(result, (result.shape[0]))
import pandas as pd
list = [*range(1, len(result)+1)]
outputDict = {"Id":list, "Category": result}
df = pd.DataFrame(outputDict)
df.to_csv('wineResult.csv', index=False)
```

```
(1000, 1)
```

[ ]:
```

```

Q4

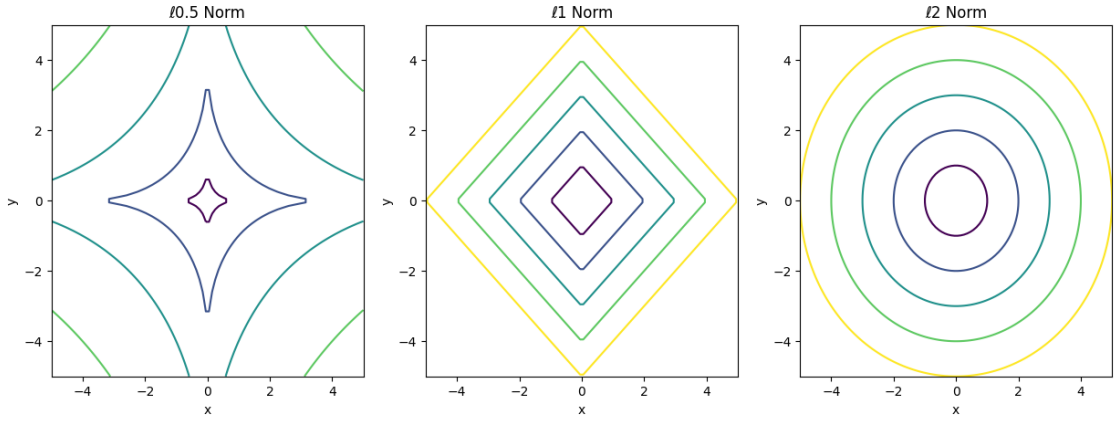
March 8, 2024

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
# Define a range for plotting
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)

# Compute norms for each point
def l0_5_norm(w):
    return np.sqrt(np.abs(w))

def l1_norm(w):
    return np.abs(w)

def l2_norm(w):
    return w**2

# Compute norms on the grid
Z_0_5 = l0_5_norm(X) + l0_5_norm(Y)
Z_1 = l1_norm(X) + l1_norm(Y)
Z_2 = np.sqrt(l2_norm(X) + l2_norm(Y))

print((Z_2).shape)

# Plot
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.contour(X, Y, Z_0_5, levels=[1, 2, 3, 4, 5])
plt.title(' 0.5 Norm')
plt.xlabel('x')
plt.ylabel('y')

plt.subplot(1, 3, 2)
plt.contour(X, Y, Z_1, levels=[1, 2, 3, 4, 5])
plt.title(' 1 Norm')
plt.xlabel('x')
```

```
plt.ylabel('y')

plt.subplot(1, 3, 3)
plt.contour(X, Y, Z_2, levels=[1, 2, 3, 4, 5])
plt.title(' 2 Norm')
plt.xlabel('x')
plt.ylabel('y')

# plt.tight_layout()
plt.show()
```

```
(100, 100)
```



[ ]: