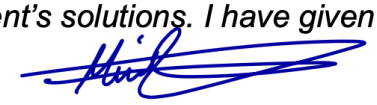
Hiva Mohammadzadeh
3036919598

Question 1: Honor Code

"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."

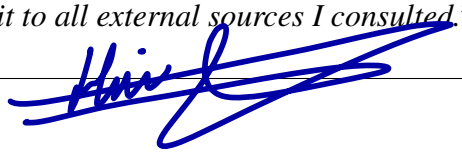


Question 2: Logistic Regression with Newton's Method

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: 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. $\mathbf{w}_i$ 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.

2) 1) Derive $\nabla_w J(w)$. Also derive all intermediate derivatives in matrix-vector form

$$J(w) = -y \cdot \ln s - (1-y) \cdot \ln(1-s)$$

$$\nabla_w J(w) = \nabla_w \left(-y \cdot \ln s - (1-y) \cdot \ln(1-s)\right) = -\nabla_w (y \cdot \ln(s)) - \nabla_w ((1-y) \cdot \ln(1-s)) \qquad \nabla_x (y \cdot z) = (\nabla_x y) z + (\nabla_x z) y$$

$$= -\nabla_w (y) \ln(s) - \nabla_w (\ln(s)) y - \nabla_w (1-y) \ln(1-s) - \nabla_w (\ln(1-s))(1-y)$$

$$= -\nabla_w (\ln(s)) y - \nabla_w (\ln(1-s))(1-y) \qquad \nabla_x f(y) = (\nabla_x y) \nabla_y f(y)$$

$$= -\nabla_w (s) \nabla_s \ln(s) y - (\nabla_w (1-s)) \nabla_s (\ln(1-s)) (1-y) = -(\nabla_w (s)) \operatorname{diag}\left(\tfrac{1}{s}\right) y - (\nabla_w (1-s)) \operatorname{diag}\left(\tfrac{1}{(1-s)}\right)(1-y)$$

$\nabla_w (Xw) \nabla_{Xw} (s(Xw)) = X^T \nabla_{Xw} (s(Xw)) = X^T \operatorname{diag}(s(1-s))$; $\nabla_w (1-s) = X^T \operatorname{diag}(s(1-s))$

$$= -X^T \underbrace{\operatorname{diag}(s(1-s))}_{\operatorname{diag}(1-s)} \operatorname{diag}(1/s) y - X^T \underbrace{\operatorname{diag}(s(1-s))}_{\operatorname{diag}(s)} \operatorname{diag}\left(1/(1-s)\right)(1-y) = -X^T \operatorname{diag}(1-s_i) y - X^T \operatorname{diag}(s_i)(1-y)$$

$$= -X^T \operatorname{diag}(s_i) 1 - X^T \operatorname{diag}(1-s_i) - X^T \operatorname{diag}(s_i) y \qquad s = \operatorname{diag}(s_i) 1$$

$$= -X^T s - X^T \underbrace{(\operatorname{diag}(1-s_i) - \operatorname{diag}(s_i))}_{I} y = -X^T s - X^T y = -X^T (s-y) = \boxed{X^T (y-s)}$$

---

2) Derive the Hessian $\nabla_w^2 J(w)$ as a matrix vector expression.

$$\nabla_w^2 J(w) = -\nabla_w (X^T (y-s)) = \underbrace{-\nabla_w (y-s)}_{-X^T \operatorname{diag}(s(1-s))} X = \underbrace{X^T \operatorname{diag}(s(1-s))}_{\Omega} X = \boxed{X^T \Omega X}$$

$$\downarrow$$

$$\operatorname{diag}(s_1(1-s_1), s_2(1-s_2), \ldots, s_n(1-s_n)) = \Omega$$

---

3) Write the matrix-vector update law for one iteration of Newton's method.

$$w_{t+1} = w_t - \left(\nabla_w^2 J(w)\right)^{-1} \nabla_w J$$

$$\boxed{= w_t - (X^T \Omega X)^{-1} (X^T (y-s))}$$

②cont

④ $x_1 = [0.2 \quad 3.1]^T$ $x_2 = [1.0 \quad 3.0]^T$ $x_3 = [-0.2 \quad 1.2]^T$ $x_4 = [1.0 \quad 1.1]^T$ and labels $y_1 = 1, y_2 = 1, y_3 = 0, y_4 = 0$. Initial weight: $w^{(0)} = [-1 \quad 1 \quad 0]^T$

a) Initial value of $s^{(0)} = ?$

$s^{(0)} = [0.9478 \quad 0.8808 \quad 0.8022 \quad 0.5249]$

b) The value of w after 1 iteration $w^{(1)} = ?$

$w^{(1)} = [1.3247 \quad 3.0499 \quad -6.8291]$

c) The value of s after 1 iteration. $s^{(1)} = ?$

$s^{(1)} = [0.9474 \quad 0.9746 \quad 0.0312 \quad 0.1044]$

d) The value of w after 2 iterations $w^{(2)} = ?$

$w^{(2)} = [1.3659 \quad 4.1575 \quad -9.1998]$

(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).

3 Wine Classification with Logistic Regression

The wine dataset `data.mat` consists of 6,497 sample points, each having 12 features. The description of these features is provided in `data.mat`. The dataset includes a training set of 6,000 sample points and a test set of 497 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 4.1.

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.

SGD is slower and converges to a higher cost/loss.

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

③ on writeup.

1) $J(w) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \ln(\sigma(x_i;w)) + (1-y)\ln(1-\sigma(x_i;w))\right] + \frac{\lambda}{2n}\|w\|^2$ → loss function

$$\nabla_w J = -\frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i}{\sigma_i}\nabla\sigma_i - \frac{1-y_i}{1-\sigma_i}\nabla\sigma_i\right) + \frac{\lambda}{n}\sum_{j=1}^{d} w_j$$

$$= -\frac{1}{n}\sum_{i=1}^{n}(y_i - \sigma_i)x_i + \frac{\lambda}{n}\sum_{j=1}^{d} w_j \quad = -x^T(y - \sigma(xw)) + \frac{\lambda}{n}\sum_{j=1}^{d} w_j$$

$$w \leftarrow w + (x^T(y - \sigma(xw)) - \frac{\lambda}{n}\sum_{j=1}^{d} w_j$$ → update rule

3)

$$w \leftarrow w + \alpha\left[(y_i - \sigma(x_i;w))x_i - \frac{\lambda}{n}\sum_{j=1}^{d} w_j\right]$$ → update rule

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

converges much faster but still not as good as BGD.

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

Plot 3.4 and 3.5.

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.

Question 3: Wine Classification with Logistic Regression

Preprocessing: Normalizing, shuffling and splitting:

```
Main data: (6000, 12)
Main labels: (6000, 1)
Main test: (497, 12)

Shapes after shuffle and split:
Main data: (5000, 12)
Main labels: (5000,)
Validation data: (1000, 12)
Validation labels:(1000,)

After normalizing training
Main data (5000, 13)
[[ 0.21238764 -0.42529608  0.28218208 ...  0.42684345 -0.92568693
   1.        ]
 [-0.4041222  -0.24219715  1.24063449 ... -1.32978738  0.21432161
   1.        ]
 [ 1.59953477  0.91742937  0.28218208 ...  0.34319436  0.21432161
   1.        ]
 ...
 [-0.24999474 -1.15769178 -0.19704412 ...  0.59414162  1.35433015
   1.        ]
 [-0.24999474 -0.79149393  0.07679942 ... -0.15870016  0.21432161
   1.        ]
 [ 0.13532391  3.81649567 -2.18240982 ... -0.24234925 -0.92568693
   1.        ]]
```

▪ ▪ ▪

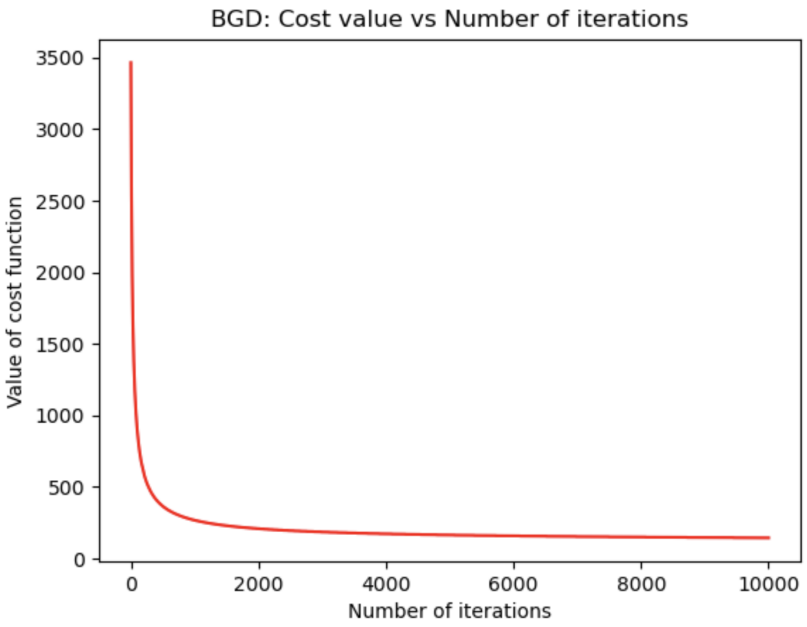
```
After normalizing test
Test data (497, 13)
[[ 0.46904252 -0.5677573   0.3362711  ...  0.77846078 -0.90726388
   1.        ]
 [ 2.13766491  0.85572333 -0.8290917  ... -1.06263446  0.26089713
   1.        ]
 [ 0.84827488  1.1522818  -0.28068568 ... -0.39314528  0.26089713
   1.        ]
 ...
 [-1.65465871  0.32191809  1.22743088 ...  2.61955602  0.26089713
   1.        ]
 [ 0.31734958  0.2626064  -0.07503342 ... -0.64420373 -0.90726388
   1.        ]
 [-0.82034751 -0.50844561  1.29598163 ... -0.56051758  0.26089713
   1.        ]]
```

Part 1: Batch Gradient Descent Update Rule:

$$w \leftarrow w + (x^T(y - \sigma(xw)) - \frac{\lambda}{n} \sum_{j=1}^{d} w_j$$ → update rule

Part 2: Batch Gradient Descent Code:

```
---- Training Batch gradient Descent---
Training accuracy: 0.9928
Validation accuracy: 0.995
```



Setting hyperparameters:

```
Setting Hyperparameters for Batch Gradient Descent:
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.537
Parameters(1e-07, 0.001, 500): Validation accuracy: 0.568
Parameters(1e-07, 0.001, 1000): Validation accuracy: 0.595
Parameters(1e-07, 0.001, 10000): Validation accuracy: 0.889
Parameters(1e-07, 0.001, 10000): Validation accuracy: 0.889
Parameters(1e-07, 0.001, 10000): Validation accuracy: 0.889
Parameters(1e-07, 0.001, 10000): Validation accuracy: 0.889
Parameters(1e-07, 0.001, 10000): Validation accuracy: 0.889
Parameters(1e-07, 0.001, 10000): Validation accuracy: 0.889
Parameters(1e-07, 0.001, 10000): Validation accuracy: 0.889
Parameters(1e-07, 0.001, 10000): Validation accuracy: 0.889
```

▪ ▪ ▪

```
Best validation accuracy:  0.992

Best hyperparameters combo:(learning rate, regularization parameter, number of iteration):  (1e-05, 0.001, 10000)
```

Part 3: Stochastic Gradient Descent (SGD) Update Rule:

3) $w \leftarrow w + \alpha [(y_i - \sigma(x_i; w)) x_i - \frac{\lambda}{n} \sum_{j=1}^{d} w_j$ → update rule

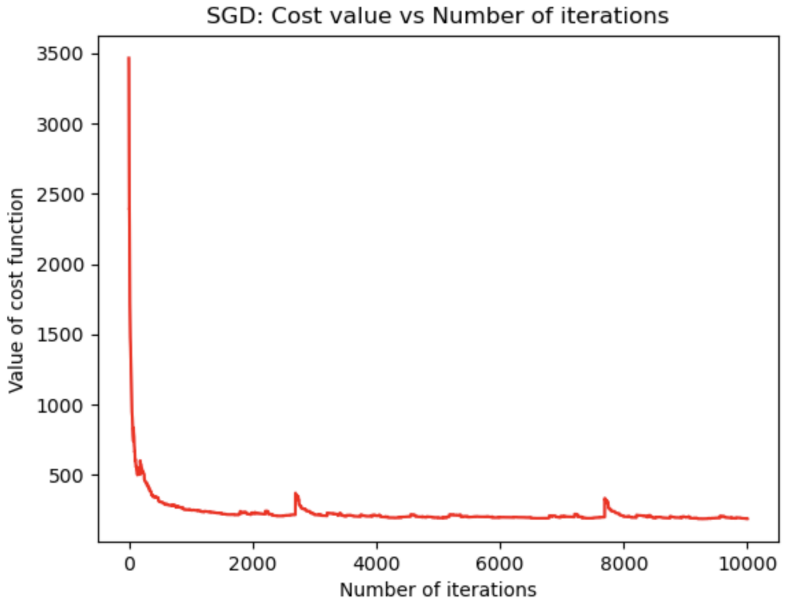
Part 4: Stochastic Gradient Descent Code:

```
---- Training Stochastic Gradient Descent ---
Training accuracy: 0.993
Validation accuracy: 0.995
```

Setting hyperparameters:

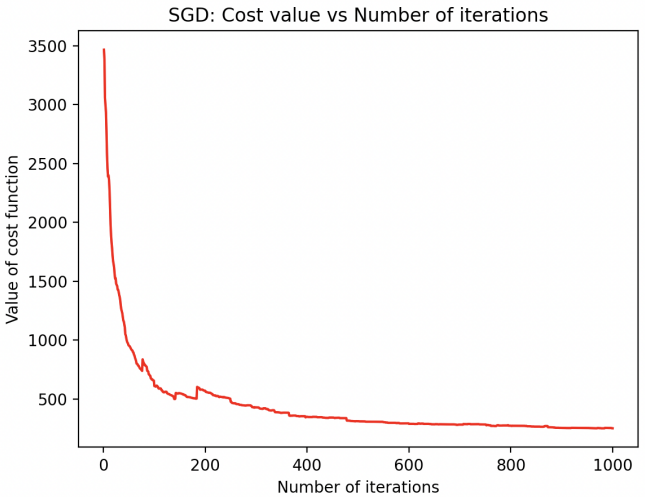
```
Setting Hyperparameters for Stochastic Gradient Descent (SGD):
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.945
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.945
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.945
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.945
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.945
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.945
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.945
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.945
Parameters(1e-07, 0.001, 100): Validation accuracy: 0.945
```

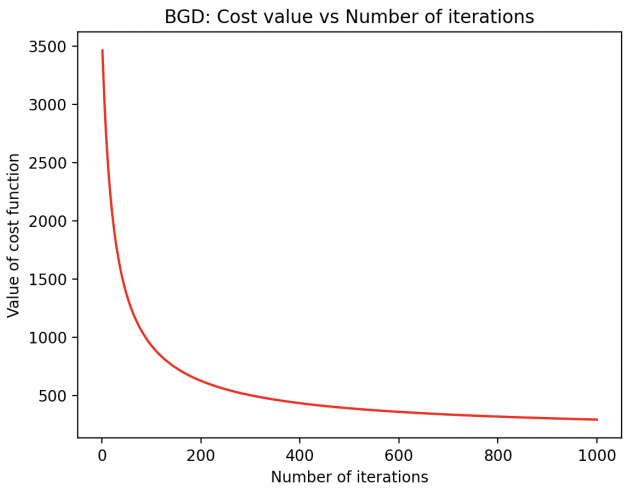
```
Parameters(0.1, 0.001, 10000): Validation accuracy: 0.992
Parameters(0.1, 0.001, 10000): Validation accuracy: 0.992
Parameters(0.1, 0.001, 10000): Validation accuracy: 0.992
Parameters(0.1, 0.001, 10000): Validation accuracy: 0.992
best validation accuracy:  0.992
best hyperparameters combo:(learning rate, regularization parameter, number of iteration):  (0.1, 0.001, 10000)
```



Compare your plot here with that of question 3.2. Which method converges more quickly?

I trained both BGD and SGD with 1000 training samples to see the difference in their convergence. As we can see Stochastic Gradient Descent converges faster. In SGD, we sample with replacement. So, we pick a random point and label every iteration.





Part 5: SGD with Decreasing Learning Rate

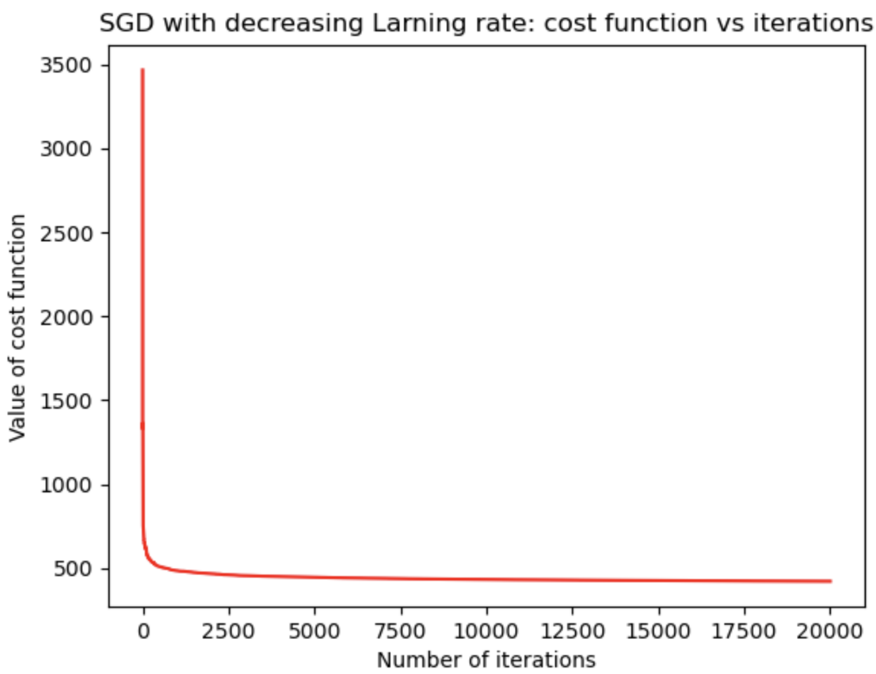
```
---- Training Stochastic Gradient Descent with decaying learning rate ---
Training accuracy: 0.9848
Validation accuracy: 0.985
```

Setting hyperparameters:

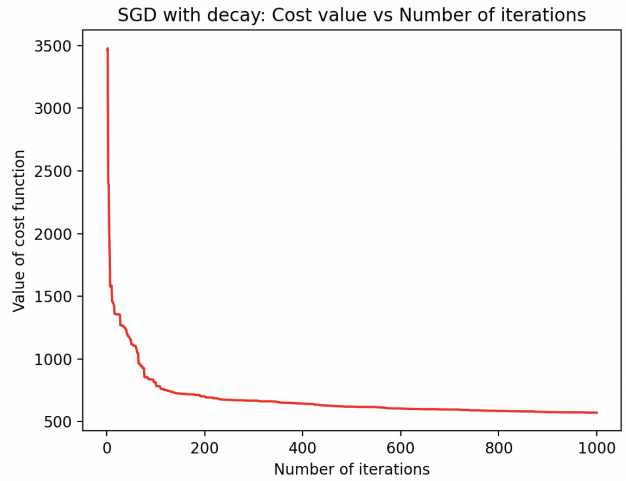
```
Setting Hyperparameters for Stochastic Gradient Descent (SGD) with decaying learning rate:
Parameters(1e-07, 0.001, 100, 0.1): Validation accuracy: 0.927
Parameters(1e-07, 0.001, 500, 0.1): Validation accuracy: 0.936
Parameters(1e-07, 0.001, 1000, 0.1): Validation accuracy: 0.941
Parameters(1e-07, 0.001, 10000, 0.1): Validation accuracy: 0.952
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
Parameters(1e-07, 0.001, 20000, 0.1): Validation accuracy: 0.954
```

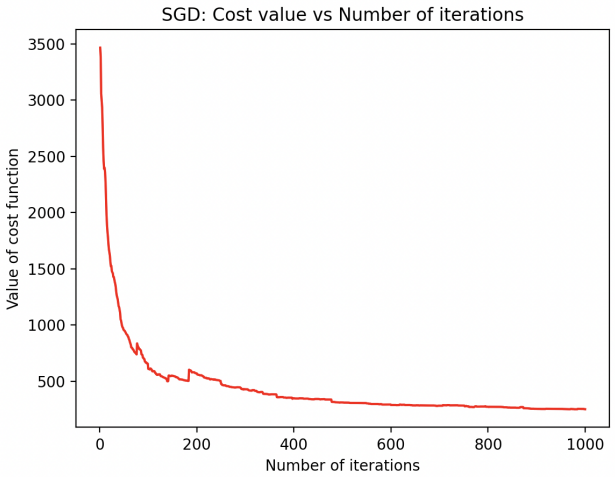
■ ■ ■

```
Parameters(1e-07, 0.01, 20000, 2): Validation accuracy: 0.985
Parameters(1e-07, 0.01, 20000, 2): Validation accuracy: 0.985
Parameters(1e-07, 0.01, 20000, 2): Validation accuracy: 0.985
Parameters(1e-07, 0.01, 20000, 2): Validation accuracy: 0.985
Parameters(1e-07, 0.01, 20000, 2): Validation accuracy: 0.985
Parameters(1e-07, 0.01, 20000, 2): Validation accuracy: 0.985
Parameters(1e-07, 0.01, 20000, 2): Validation accuracy: 0.985
best validation accuracy:  0.985
best hyperparameters combo:(learning rate, regularization parameter, number of iteration, delta):  (1e-07, 0.01, 20000, 2)
---- Training Stochastic Gradient Descent with decaying learning rate ---
Training accuracy: 0.9848
Validation accuracy: 0.985
```



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





I trained both SGD and SGD with decaying learning rate with 1000 training samples to see the difference in their convergence. As we can see SGD with decaying learning rate converges faster. The delta decreases the learning rate allowing it to converge more quickly.

Part 6: Kaggle Submission

```
---- Training Best Performing Batch gradient Descent for testing---
Tested the data and Saved the predictions
```

Kaggle username: **Hiva Mohammadzadeh**

Kaggle Scores:

a) WINE: Score: 0.97580 = 97.6%

Briefly describe what your best classifier does to achieve that score.

I did hyperparameter tuning of learning rate, regularization parameter and the number of training iterations (I tuned the number of training iterations because I was getting different accuracies when I was using different ones so I just thought I tune it to the best one.) by just having 3 for loops (4 in the case of SGD with decaying learning rate) and took the combination that gave the highest validation accuracy. Then I trained the model that had the highest validation accuracy on the entire dataset and tested it on the test set.

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.

④

1) Derive the posterior probability density law $f(w|(x_i,y_i)_{i\in[n]})$:

$$f(y_i|x_i,w) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - w\cdot x_i)^2}{2\sigma^2}} \qquad f(w|x,y) = \frac{f(w,y|x)}{f(y|x)}$$

want $f(w|(x_i,y_i)_{i\in[n]}) = f(w|(X,Y))$

$$f(w|(X,Y)) = f(w,(X,Y)) = f(Y|X,w)f(w) = \prod_{i=1}^{n} \frac{f(w,y_i|x_i)}{f(y_i|x_i)} = \prod_{i=1}^{n} \frac{f(y|x,w)f(w)}{f(y|x_i)}$$

↳ constant. y does not depend on w

$$\frac{f(y|x,w)f(w)}{f(y|x_i)} \propto \prod_{i=1}^{n} f(y_i|w,x_i) \prod_{i=1}^{n} f(w_i)$$

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

---

2) Define the log-likelihood for MAP as $\mathcal{L}(w) \triangleq \ln f(w|x_{i\in[n]}, y_{i\in[n]})$.

$$\mathcal{L}(\ln) = \ln(f(w|(X,Y)) = \Sigma\left(-\frac{(y_i - w\cdot x_i)^2}{2\sigma^2} - \frac{||w||_1}{b}\right)$$

$$\mathcal{L} = \Sigma\left(-(y_i - w\cdot x_i)^2 - (2\sigma^2)\frac{||w||_1}{b}\right) \qquad \lambda = (2\sigma^2)/b$$

$$\mathcal{L} = \Sigma(-(y_i - w\cdot x_i)^2 - \lambda||w||_1)$$

Show that minimizing the MAP log-likelihood over all w is the same as minimizing $\Sigma_{i=1}^{n}(y_i - w\cdot x_i)^2 + \lambda||w||_1$ where $||w||_1 = \Sigma_{j=1}^{d}|w_j|$

$$Map = \operatorname{argmin}(-\mathcal{L}) = \operatorname{argmin}\left(\Sigma\left((y_i - w\cdot x_i)^2 + \lambda||w||_1\right)\right)$$

$$= \boxed{\operatorname{argmin}_w\left(\Sigma_{i=1}^{n}(y_i - w\cdot x_i)^2 + \lambda||w||_1\right)}$$

Give a formula for λ as a function of the distribution parameters.

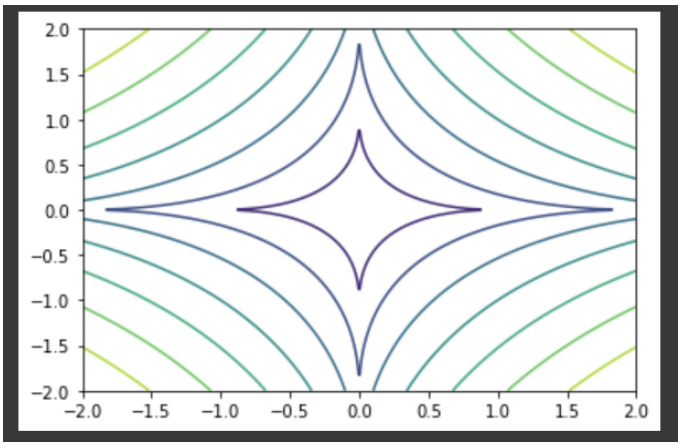
$$\boxed{\lambda = 2\sigma^2/b}$$

Question 4: A Bayesian Interpretation of Lasso

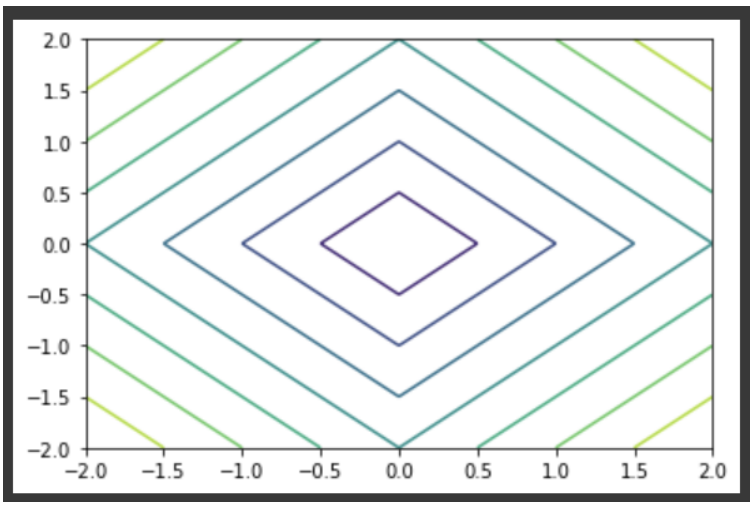
Question 5: L_1- regularization, L_2- regularization, and Sparsity

Part 1:

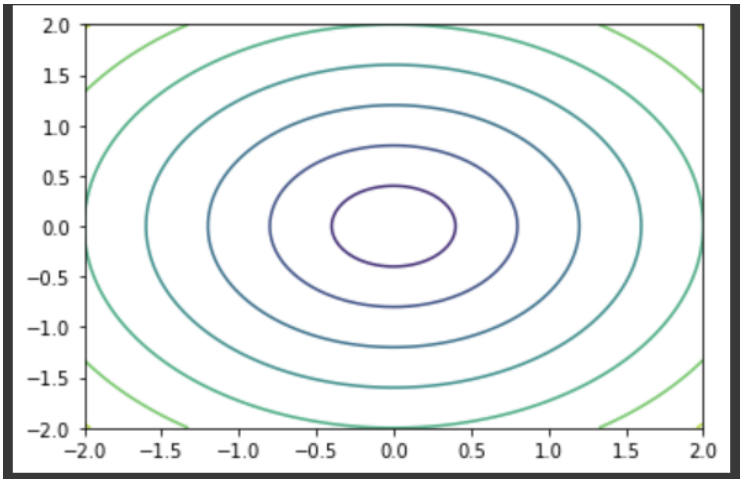
a) L_0.5



b) L_1



c) L_2



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 $\mathbf{y} \triangleq [y_1 \ \ldots \ y_n]^\top$. 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.

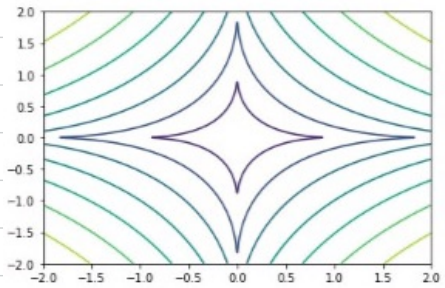
3. Using your solution to part 2, derive necessary and sufficient conditions for the i^{th} component of the optimizer $\mathbf{w}^*$ of $J_1(\cdot)$ to satisfy each of these three properties: $w_i^* > 0, w_i^* = 0$, and $w_i^* < 0$.

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 $\mathbf{w}^\#$.

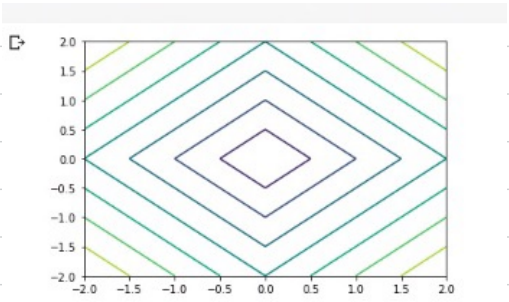
5. A vector is called *sparse* if most of its components are 0. From your solution to part 3 and 4, which of $\mathbf{w}^*$ and $\mathbf{w}^\#$ is more likely to be sparse? Why?

5) 1) Plot the iso contours with $w \in R^2$ for the following norms.

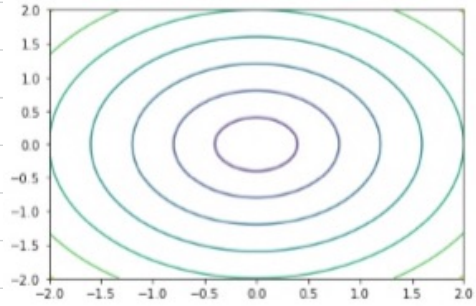
a) $l_{0.5}$



b) l_1



c) l_2



2) Show that the cost functn for l_1 least squares, $J_1(w) \triangleq \|Xw - y\|^2 + \lambda \|w\|_1$, can be written as

$J_1(w) = \|y\|^2 + \sum_{i=1}^{d} f(X_{*i}, w_i)$

$J_1(w) = \|Xw - y\|^2 + \lambda \|w\|_1 = (Xw - y)^T (Xw - y) + \lambda \sum_{i=1}^{d} w_i = w^T X^T X w - y^T X w - w^T X^T y + \|y\|^2 + \lambda \sum_{i=1}^{d} |w_i|$

$= \|y\|^2 + n \sum_{i=1}^{d} w_i^2 - 2 y^T X w + \lambda \sum_{i=1}^{d} w_i$

$Xw = X_{*1} w_1 + X_{*2} w_2 + \dots + X_{*d} w_d \rightarrow y^T X w = y^T (X_{*1} w_1 + \dots + X_{*d} w_d)$

$J_1(w) = \|y\|^2 + \sum_{i}^{d} f(X_{*i}, w_i), \quad f(X_{*i}, w_i) = -2 y^T (X_{*i} w_i) + n w_i^2 + \lambda |w_i|$

3) Using part 2, derive conditions for the ith component of the optimizer w^* of $J_1(\cdot)$ to satisfy: $w_i^* > 0$,

$w_i^* = 0$, and $w_i^* < 0$ $\quad f(0) = 0$ $\quad \min_{w_i > 0} f(w_i) < \inf_{w_i \le 0} f(w_i)$ $\quad \min_{w_i > 0} J(w_i) < 0$

$\le 0 \rightarrow$ because $f(w_i)$ is convex

optimizer w^* when $\frac{d J_1(w)}{d w_i} = 0$ $\quad f(X_{*i}, w_i) = \|X_{*i} w_i\|^2 - 2 y^T (X_{*i} w_i) + \lambda |w_i|$ ($2 y^T (X_{*i} w_i)$: fixed) $\quad$ f is convex we can optimize by $\frac{df}{dw} = 0$

when $w_i^* > 0$: $\underset{w_i > 0}{\operatorname{argmin}} f_1(w_i) = 2 n w_i + \lambda - 2 y^T X_{*i} = 0$

$\min_{w_i > 0} = -1 \{ 2 y^T X_{*i} - \lambda > 0 \} \frac{(\lambda - 2 y^T X_{*i})^2}{4n} \rightarrow w_i^* > 0$ if $\lambda - 2 y^T X_{*i} > 0$

$\{ 2 y^T X_{*i} - \lambda > 0 \}$: condition

⑤ cont.

③ cont. when $w_i^* < 0$ $\min_{w_i \le 0} f(w_i)$

So, $\operatorname{argmin}_{w_i<0} f(w_i) = 2n w_i - \lambda - 2y^T x_{*i} = 0 \rightarrow \operatorname{argmin}_{w_i<0} J(w_i) = \frac{\lambda + 2y^T x_{*i}}{2n}$

$\min_{w_i^*<0} = -1\{\lambda + 2y^T x_{*i} < 0\} \frac{(\lambda + 2y^T x_{*i})^2}{4n} \rightarrow w_i^* < 0$ if $\lambda + 2y^T x_i^* < 0$

Conditions
- $w_i^* > 0$, $\lambda - 2y^T x_i > 0$
- $w_i^* < 0$, $\lambda + 2y^T x_i < 0$
- $w_i^* = 0$, otherwise

---

④ For the optimizer $w^\#$ of the ℓ_2-least squares cost function $J_2(w) \triangleq \|Xw - y\|^2 + \lambda \|w\|^2$, derive the condition for $w_i^\# = 0$.

$J_2(w) = \|y\|^2 + \sum_{i=1}^{d} g(x_i, w_i)$ where $g(x_i, w_i) = -2y^T(x_i w_i) + n w_i^2 + \lambda w_i^2$ (convex)

$\frac{dg(w_i)}{dw_i^\#} g(w_i) = 0 \rightarrow \min \rightarrow 2(\lambda + n) w_i - 2y^T x_i^\# = 0 \rightarrow w_i^\# = \frac{y^T x_i^\#}{\lambda + n}$ if $y^T x_i = 0$

---

⑤ Using part 3 and 4, which of w^* and $w^\#$ is more likely to be sparse? Why?

Question 3. w_i^* is more likely to parse because it components are more likely to be 0 since $y^T x_i$ can be anywhere between $\frac{\lambda}{2}$ and $-\frac{\lambda}{2}$ which is a larger range than only when $y^T x_i$ is zero.

$w_i^\# = 0$ if $y^T x_i = 0$

$w_i^* = 0$ if $\frac{\lambda}{2} \ge y^T x_i \ge \frac{-\lambda}{2}$

CODE APPENDIX:

Question 3: Wine Classification with Logistic Regression

Random Seed = 100

Preprocessing: Normalizing, shuffling and splitting:

```
import numpy as np
import scipy.io as sio
import scipy
import matplotlib.pyplot as plt
from save_csv import results_to_csv
from scipy.special import *
import math

# Random seed
np.random.seed(100)

data = sio.loadmat('data.mat')
# print(data)

train_data = data['X']
print("Main data: {}" .format(train_data.shape))
train_labels = data['y']
print("Main labels: {}" .format(train_labels.shape))
test_data = data['X_test']
print("Main test: {}" .format(test_data.shape))

# Shuffle the data and split and set aside 1000 samples aside for the validation
full_set = np.concatenate((train_data, train_labels), axis=1)
np.random.shuffle(full_set)

training_data_full = full_set[:5000, :-1]
training_labels = full_set[:5000, -1:].reshape(-1,)

validation_data_full = full_set[5000:, :-1]
validation_labels = full_set[5000:, -1:].reshape(-1,)

print("\nShapes after shuffle and split:")
print("Main data: {}" .format(training_data_full.shape))
print("Main labels: {}" .format(training_labels.shape))
print("Validation data: {}" .format(validation_data_full.shape))
print("Validation labels:{}" .format(validation_labels.shape))

# 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.
```

```
# Function to normalize the training and test data with each feature's mean and standard
deviation
mean_f = None
std_f = None
def normalize(data, mean=None, standard_deviation=None):
   samples, features = data.shape
   if not mean:
       mean = data.mean(axis=0)
       standard_deviation = data.std(axis=0)
   data = (data - mean) / standard_deviation
   # Add a fictitious dimension by adding a vector of 1s at the end of the training data set.
   data = np.concatenate((data, np.ones((samples, 1))), axis=1)
   return data
# Normalize the training and test data
training_data = normalize(training_data_full)
print("\nAfter normalizing training")
print("Main data {}" .format(training_data.shape))
print(training_data)

test_data = normalize(test_data)
print("\nAfter normalizing test")
print("Test data {}" .format(test_data.shape))
print(test_data)
```

Part 2: Batch Gradient Descent Code:

```
#### QUESTION 3.2: Batch Gradient Descent Code.

# 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.

# Random seed = 100

def sigmoid_func(X, w):

   # print('hi')

   # print(X.shape)

   # # print(w.shape)

   # print(w.shape)

   return scipy.special.expit(np.matmul(X, w.T))

# Loss function of the logistic regression for BGD

def loss_func(data, label, weight, reg):

   regularizer = (reg / 2) * weight.T.dot(weight)

   # w = np.reshape(w, (w.shape[0], 1))
```

```
    # print(X.shape)
    # print(w.shape)
    loss_1 = np.dot(label.T, np.log(sigmoid_func(data, weight)))
    loss_2 = np.dot((1 - label).T, np.log(1 - sigmoid_func(data, weight)))
    # print(regularizer - loss_1 - loss_2)
    return regularizer - loss_1 - loss_2
# Update rule of BGD
def update_rule(data, label, weight, reg, lr):
    # print('Hoala')
    # print(X.T.shape)
    # print(y.shape)
    sigmoid = sigmoid_func(data, weight)
    # print('bro')
    # print((y - sigmoind.T).shape)
    # print(X.shape)
    gradient = np.matmul(data.T, (label - sigmoid))
    # print(gradient.shape)
    # print('hi')
    # print(w.shape)
    # w = np.reshape(w, (13,1))
    # print(w.shape)
    step = reg * weight - gradient
    return weight - (lr * step)
X_train = normalize(training_data_full)
# Function to train the BGD Logistic Regression
def train_BGD(num_iterations, X_train, y_train, w_train, regularization_param, lr):
    training_loss = []
    for iteration in range(num_iterations):
        loss = loss_func(X_train, y_train, w_train, regularization_param)
        next_w = update_rule(X_train, y_train, w_train, regularization_param, lr)
        # print(loss)
        if math.isnan(loss):
            loss = 0
        training_loss.append(int(loss))
        w_train = next_w
    return w_train, training_loss
print("\nSetting Hyperparameters for Batch Gradient Descent:")
```

```
### Setting the hyperparameters of the BGD.

w_train = np.zeros(13)

num_iterations = [100, 500, 1000, 10000]

regularization_param = [0.001, 0.01, 0.1, 1.0, 10]

learningRates = [1e-7, 1e-6,1e-5, 1e-3, 0.1, 1, 10]

best_validation_accuracy_so_far = 0.0

best_hyperparameters_so_far = None

for lr in learningRates:

   for l in regularization_param:

       for i in num_iterations:

           batch_best_w, batch_tl = train_BGD(i, X_train, training_labels, w_train, l, lr)

           val_data = normalize(validation_data_full, mean_f, std_f)

           out_labels = sigmoid_func(val_data, batch_best_w)

           predicted_labels = (out_labels > 0.5).astype(np.int32)

           predicted_labels = predicted_labels.reshape(-1,)

           val_accuracy = (validation_labels == predicted_labels).astype(np.int32).mean()

           if val_accuracy > best_validation_accuracy_so_far:

               best_validation_accuracy_so_far = val_accuracy

               best_hyperparameters_so_far = (lr, l, i)

           print("Parameters{}: Validation accuracy: {}" .format(best_hyperparameters_so_far,
best_validation_accuracy_so_far))



best_validation_acc = best_validation_accuracy_so_far

best_hyperparams = best_hyperparameters_so_far

print("\nBest validation accuracy: ",best_validation_acc)

print("\nBest hyperparameters combo:(learning rate, regularization parameter, number of
iteration): ",best_hyperparams)

## Training BGD

print("\n---- Training Batch gradient Descent---")

w_train = np.zeros(13)

# Were set by hyperparameter tuning above

num_iterations = 10000

regularization_param = 0.001

lerningRate = 1e-5

batch_GD_best_w, batch_trainingloss = train_BGD(num_iterations, X_train, training_labels,
w_train, regularization_param, lerningRate)

# Plot the value of the cost function versus the number of iterations spent in training.

iterations = list(np.arange(1, num_iterations + 1, 1))

plt.figure(1)
```

```
plt.plot(iterations, batch_trainingloss, 'r-')

plt.title("BGD: Cost value vs Number of iterations")

plt.xlabel("Number of iterations")

plt.ylabel("Value of cost function")

plt.savefig('Batch Gradient Descent.png')

# plt.show()

out_labels = sigmoid_func(X_train, batch_GD_best_w)

predicted_labels = (out_labels > 0.5).astype(np.int32)

predicted_labels = predicted_labels.reshape(-1,)

(training_labels == predicted_labels).astype(np.int32).mean()

print("Training accuracy: {}" .format((training_labels ==
predicted_labels).astype(np.int32).mean()))

val_data = normalize(validation_data_full, mean_f, std_f)

out_labels = sigmoid_func(val_data, batch_GD_best_w)

predicted_labels = (out_labels > 0.5).astype(np.int32)

predicted_labels = predicted_labels.reshape(-1,)

(validation_labels == predicted_labels).astype(np.int32).mean()

print("Validation accuracy: {}" .format((validation_labels ==
predicted_labels).astype(np.int32).mean()))
```

Part 4: Stochastic Gradient Descent Code

```
#### QUESTION 3.4 : Stochastic Gradient Descent Code.

# 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.

def sigmoid_single(x_i, w):

    return 1 / (1 + np.exp(-np.dot(x_i, w)))

# Update rule for SGD

def sgd_compute_update(x_i, y_i, w, reg, lr):

    sig = sigmoid_single(x_i.T,w)

    gradient = np.dot(x_i, (y_i-sig))

    step = reg * w - gradient

    return w - (lr * step)

X_train = normalize(training_data_full)

# Function to train the SGD Logistic Regression
```

```
def train_SGD(num_iterations, X_train, y_train, w_train, regularization_param, lr):

   training_loss_history = []

   N, d = X_train.shape

   for curr_iter in range(num_iterations):

       training_sample = X_train[curr_iter % N]

       labels_sample = y_train[curr_iter % N]

       loss = loss_func(X_train, y_train, w_train, regularization_param)

       w_train = sgd_compute_update(training_sample, labels_sample, w_train,
regularization_param, lr)

       training_loss_history.append(float(loss))



   return w_train, training_loss_history

print("Setting Hyperparameters for Stochastic Gradient Descent (SGD):")

### Setting the hyperparameters of the best performed model. Which is the  Stochastic Gradient
Descent (SGD)

w_train = np.zeros(13)

num_iterations = [100, 500, 1000, 10000]

regularization_param = [0.001, 0.01, 0.1, 1.0, 10]

learningRates = [1e-7, 1e-6, 1e-5, 1e-3, 0.1, 1, 10]

best_validation_accuracy_so_far = 0.0

best_hyperparameters_so_far = None

for a in learningRates:

   for l in regularization_param:

       for i in num_iterations:

           batch_best_w, batch_tl = train_SGD(i, X_train, training_labels, w_train, l, a)

           data_val = normalize(validation_data_full, mean_f, std_f)

           out_labels = sigmoid_func(data_val, batch_best_w)

           pred_label = (out_labels > 0.5).astype(np.int32)

           pred_label= pred_label.reshape(-1,)

           val_accuracy = (validation_labels == pred_label).astype(np.int32).mean()

           if val_accuracy > best_validation_accuracy_so_far:

               best_validation_accuracy_so_far = val_accuracy

               best_hyperparameters_so_far = (a, l, i)

           print("Parameters{}: Validation accuracy: {}" .format(best_hyperparameters_so_far,
best_validation_accuracy_so_far))

best_valudation_acc = best_validation_accuracy_so_far

best_hyperparams = best_hyperparameters_so_far

print("best validation accuracy: ",best_valudation_acc)
```

```
print("best hyperparameters combo:(learning rate, regularization parameter, number of
iteration): ",best_hyperparams)

## Training SGD

print("---- Training Stochastic Gradient Descent ---")

w_train = np.zeros(13)

# Set by hyperparameter tuning

regularization_param = 0.001

lr = 0.1

num_iterations = 10000

sgd_best_w, sgd_trainingloss = train_SGD(num_iterations, X_train, training_labels, w_train,
regularization_param, lr)

iterations = list(np.arange(1, num_iterations + 1, 1))

plt.figure(2)

plt.plot(iterations, sgd_trainingloss, 'r-')

plt.title("SGD: Cost value vs Number of iterations")

plt.xlabel("Number of iterations")

plt.ylabel("Value of cost function")

plt.savefig('Stochastic Gradient Descent.png')

# plt.show()

out_labels = sigmoid_func(X_train, sgd_best_w)

predicted_labels = (out_labels > 0.5).astype(np.int32)

predicted_labels= predicted_labels.reshape(-1,)

(training_labels == predicted_labels).astype(np.int32).mean()

print("Training accuracy: {}" .format((training_labels ==
predicted_labels).astype(np.int32).mean()))

out_labels = normalize(validation_data_full, mean_f, std_f)

out_labels = sigmoid_func(out_labels, sgd_best_w)

predicted_labels = (out_labels > 0.5).astype(np.int32)

predicted_labels = predicted_labels.reshape(-1,)

(validation_labels == predicted_labels).astype(np.int32).mean()

print("Validation accuracy: {}" .format((validation_labels ==
predicted_labels).astype(np.int32).mean()))
```

Part 5: Stochastic Gradient Descent with decaying learning rate

```
#### QUESTION 3-5: Stochastic Gradient Descent with decaying learning rate

# 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 εt = δ/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?

X_train = normalize(training_data_full)

# Function to train the SGD Logistic Regression with decaying learning rate.

def train_sgd_decay(num_iterations, X_train, y_train, w_train, lr, regularization_param,
delta):

   training_loss_history = []

   N, d = X_train.shape

   for iteration in range(num_iterations):

       lr = delta / (iteration + 1)

       X_sample = X_train[iteration % N]

       y_sample = y_train[iteration % N]

       loss = loss_func(X_train, y_train, w_train, regularization_param)

       w_train = sgd_compute_update(X_sample, y_sample, w_train, regularization_param, lr)

       training_loss_history.append(float(loss))

   return w_train, training_loss_history



print("\nSetting Hyperparameters for Stochastic Gradient Descent (SGD) with decaying learning
rate:")

### Setting the hyperparameters of the best performed model. Which is the  Stochastic Gradient
Descent (SGD)

w_train = np.zeros(13)

deltas = [0.1, 0.2, 0.5, 0.7, 0.9, 1.0, 2, 5]

num_iterations = [100, 500, 1000, 10000, 20000]

regularization_param = [ 0.001, 0.01, 0.1, 1.0, 10]

learningRates = [1e-7, 1e-6, 1e-5, 1e-3, 0.1, 1, 10]

best_validation_accuracy_so_far = 0.0

best_hyperparameters_so_far = None

for d in deltas:

   for a in learningRates:

       for l in regularization_param:

           for i in num_iterations:

               batch_best_w, batch_tl = train_sgd_decay(i, X_train, training_labels, w_train,
lr, l, d)

               val_data = normalize(validation_data_full, mean_f, std_f)

               out_labels = sigmoid_func(val_data, batch_best_w)

               predicted_labels = (out_labels > 0.5).astype(np.int32)

               predicted_labels = predicted_labels.reshape(-1,)

               val_accuracy = (validation_labels == predicted_labels).astype(np.int32).mean()
```

```
                if val_accuracy > best_validation_accuracy_so_far:

                    best_validation_accuracy_so_far = val_accuracy

                    best_hyperparameters_so_far = (a, l, i, d)

                print("Parameters{}: Validation accuracy: {}"
.format(best_hyperparameters_so_far, best_validation_accuracy_so_far))



best_val_acc = best_validation_accuracy_so_far

best_hyperparams = best_hyperparameters_so_far

print("best validation accuracy: ",best_val_acc)

print("best hyperparameters combo:(learning rate, regularization parameter, number of
iteration, delta): ",best_hyperparams)

## Training SGD with decaying learning rate


print("---- Training Stochastic Gradient Descent with decaying learning rate ---")

w_train = np.zeros(13)

# Set by hyperparameter tuning

regularization_param = 0.01

lr = 1e-07

delta = 2.0

num_iterations = 20000

sgd_decay_best_w, sgd_decay_tl = train_sgd_decay(num_iterations, X_train, training_labels,
w_train, lr, regularization_param, delta)



iterations = list(np.arange(1, num_iterations+1, 1))

plt.figure(3)

plt.plot(iterations, sgd_decay_tl, 'r-')

plt.title("SGD with decreasing Learning rate: cost function vs iterations")

plt.xlabel("Number of iterations")

plt.ylabel("Value of cost function")

plt.savefig('SGD with decreasing learning rate.png')

# plt.show()

out_labels = sigmoid_func(X_train, sgd_decay_best_w)

predicted_labels = (out_labels > 0.5).astype(np.int32)

predicted_labels = predicted_labels.reshape(-1,)

(training_labels == predicted_labels).astype(np.int32).mean()

print("Training accuracy: {}" .format((training_labels ==
predicted_labels).astype(np.int32).mean()))

out_labels = normalize(validation_data_full, mean_f, std_f)

out_labels = sigmoid_func(out_labels, sgd_decay_best_w)

predicted_labels = (out_labels > 0.5).astype(np.int32)
```

```
predicted_labels = predicted_labels.reshape(-1,)

(validation_labels == predicted_labels).astype(np.int32).mean()

print("Validation accuracy: {}" .format((validation_labels ==
predicted_labels).astype(np.int32).mean()))
```

Part 6: Kaggle

```
#### QUESTION 3.6: Kaggle

# 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.

## My Best classifier is the batch gradient Descent classifier which got 99.5% validation
accuracy

## Training with the whole training set

X_train = normalize(training_data_full)

print("\n---- Training Best Performing Batch gradient Descent for testing---")

w_train = np.zeros(13)

#set by hyperparameter tuning

num_iterations = 10000

regularization_param = 0.001

lr = 1e-5

batch_best_w, batch_train_loss = train_BGD(num_iterations, X_train, training_labels, w_train,
regularization_param, lr)

test_data_full = data["X_test"]

X_test = normalize(test_data_full, mean_f, std_f)

out_labels = sigmoid_func(X_test, batch_best_w)

predicted_labels = (out_labels > 0.5).astype(np.int32)

predicted_labels = predicted_labels.reshape(-1,)

results_to_csv(predicted_labels)

print("Tested the data and Saved the predictions")
```

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.