CS 189 Introduction to Machine Learning
Spring 2023 Jonathan Shewchuk

HW4

Due: Friday, March 10 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/spring23-cs189-hw4-wine/
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.
- 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: _____

Solution: [RUBRIC: (+1 point) if declared and signed.]

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.*

Solution: [**RUBRIC**: Total (+11 points) for Q2.]

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 $\mathbf{w}$).

Solution:

[**RUBRIC**: Throughout (2.1), do NOT accept any intermediate or final answer which is not written as expression of vectors and/or matrices.]

Let X be the design matrix (whose rows are $\mathbf{x}_i^\top$). Since $\mathbf{y}$ and $\mathbf{1} - \mathbf{y}$ are independent of $\mathbf{w}$, use $\nabla_{\mathbf{x}} \big(\mathbf{y} \cdot \mathbf{z}\big) = \big(\nabla_{\mathbf{x}} \mathbf{y}\big)\mathbf{z} + \big(\nabla_{\mathbf{x}} \mathbf{z}\big)\mathbf{y} = \big(\nabla_{\mathbf{x}} \mathbf{z}\big)\mathbf{y}$ to get

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = -\Big(\nabla_{\mathbf{w}} \ln \mathbf{s}(X\mathbf{w})\Big)\mathbf{y} - \Big(\nabla_{\mathbf{w}} \ln(\mathbf{1} - \mathbf{s}(X\mathbf{w}))\Big)(\mathbf{1} - \mathbf{y}) \tag{1}$$

[**RUBRIC**: (+1 point) for correct application of identity. (+0.5 point) for partially correct application of identity.]

Now apply the chain rule on $\nabla_{\mathbf{w}} \ln \mathbf{s}(X\mathbf{w})$ and $\nabla_{\mathbf{w}} \ln\big(\mathbf{1} - \mathbf{s}(X\mathbf{w})\big)$ to get

$$\begin{aligned} \nabla_{\mathbf{w}} \ln \mathbf{s}(X\mathbf{w}) &= \big(\nabla_{\mathbf{w}} X\mathbf{w}\big)\big(\nabla_{\mathbf{x}} \mathbf{s}(\mathbf{x})\big)\big(\nabla_{\mathbf{s}} \ln \mathbf{s}\big) \\ \nabla_{\mathbf{w}} \ln\big(\mathbf{1} - \mathbf{s}(X\mathbf{w})\big) &= \big(\nabla_{\mathbf{w}} X\mathbf{w}\big)\big(\nabla_{\mathbf{x}} \mathbf{s}(\mathbf{x})\big)\big(\nabla_{\mathbf{s}} \ln(\mathbf{1} - \mathbf{s})\big) \end{aligned} \tag{2}$$

[**RUBRIC**: (+1 point) for correct application of chain rule. (+0.5 point) for partially correct application of chain rule.]

Thus, observe that $\nabla_{\mathbf{w}} X\mathbf{w} = X^\top$, $\nabla_{\mathbf{x}} \mathbf{s}(\mathbf{x}) = \text{diag}(s_i(1-s_i))$ where $\text{diag}(a_i)$ denotes the diagonal matrix with entries a_i, $\nabla_{\mathbf{s}} \ln \mathbf{s} = \text{diag}(1/s_i)$ and $\nabla_{\mathbf{s}} \ln(\mathbf{1} - \mathbf{s}) = \text{diag}\big(-\frac{1}{1-s_i}\big)$.

[**RUBRIC**: For each correct intermediate derivative, add (+0.5 points) for a correct derivation, and (+0.5 points) for a correct matrix-vector answer. You should accept a component-wise derivation but the final expression of these intermediate derivatives *must be* in matrix-vector form. Thus, total $4 \times (0.5 + 0.5) =$ (+4 points) for correct derivations and answers of 4 intermediates above.]

Substitution into (2) gives $\nabla_{\mathbf{w}} \ln \mathbf{s}(X\mathbf{w}) = X^\top \text{diag}(s_i(1 - s_i)) \text{diag}(1/s_i) = X^\top \text{diag}(1 - s_i)$.

Substitution into (2) gives $\nabla_{\mathbf{w}} \ln(\mathbf{1} - \mathbf{s}(X\mathbf{w})) = X^\top \text{diag}(s_i(1 - s_i)) \text{diag}\big(-\frac{1}{1-s_i}\big) = -X^\top \text{diag}(s_i)$.

Substitute these into (1) and simplify, giving

$$\begin{aligned} \nabla_{\mathbf{w}} J(\mathbf{w}) &= -X^\top \text{diag}(1 - s_i)\mathbf{y} + X^\top \text{diag}(s_i)(\mathbf{1} - \mathbf{y}) \\ &= X^\top \text{diag}(s_i)\mathbf{1} - X^\top \text{diag}(1 - s_i)\mathbf{y} - X^\top \text{diag}(s_i)\mathbf{y} \\ &= X^\top \mathbf{s} - X^\top \mathbf{y} \quad \text{(using } \text{diag}(s_i)\mathbf{1} = \mathbf{s} \text{ and } \text{diag}(1 - s_i) + \text{diag}(s_i) = I\text{)} \\ &= X^\top(\mathbf{s} - \mathbf{y}) \end{aligned} \tag{3}$$

[**RUBRIC**: (+1 point) for correct final answer.]

[**RUBRIC**: Total (+7 points) for 2.1!]

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

Solution: Begin from solution of sub-part (1.1) to get $\nabla_{\mathbf{w}}(\nabla_{\mathbf{w}} J) = \nabla_{\mathbf{w}} X^\top(\mathbf{s} - \mathbf{y}) = \nabla_{\mathbf{w}} X^\top \mathbf{s}$. Since X does not depend on $\mathbf{w}$, use identity $\nabla_{\mathbf{x}} C\mathbf{y}(\mathbf{x}) = \big(\nabla_{\mathbf{x}} \mathbf{y}(\mathbf{x})\big) C^\top$ to get $\nabla^2_{\mathbf{w}} J = \big(\nabla_{\mathbf{w}} \mathbf{s}(X\mathbf{w})\big) X$.

Also from sub-part (1.1) we know $\nabla_{\mathbf{w}} \mathbf{s}(X\mathbf{w}) = X^\top \text{diag}(s_i(1 - s_i))$. Define $\Omega \triangleq \text{diag}(s_i(1 - s_i))$ to finally get $\nabla^2_{\mathbf{w}} J = X^\top \Omega X$.

[**RUBRIC**: (+0.5 point) for correct substitution. (+0.5 point) for correct Hessian.]

[**RUBRIC**: Total (+1 point) for 2.2!]

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

Solution: The update law for Newton's method is $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - (\nabla^2_{\mathbf{w}} J)^{-1}(\nabla_{\mathbf{w}} J)$. Subsitution from sub-parts (1.1) and (1.2) gives $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + (X^\top \Omega X)^{-1} X^\top(\mathbf{y} - \mathbf{s})$

[**RUBRIC**: (+1 point) for correct update law. Some students might express the process of applying $(X^\top \Omega X)^{-1}$ as solving a linear system. That is also considered correct.]

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.

Solution:

```
import numpy as np

# Helper func
def sigmoid(x):
    sigm = 1. / (1. + np.exp(-x))
    return sigm

def calc_next_si(X, wi):
  return sigmoid(X.dot(wi))

def calc_next_wi(X, wi, si, lambda_):
  Omega = np.diag(np.multiply(si, 1 - si))
  XtOX_term = X.T.dot(Omega).dot(X)
  left_side = np.linalg.inv((2 * lambda_ * np.eye(XtOX_term.shape[0]) + XtOX_term))

  #print the inverse
  #print(left_side)

  right_side = 2 * lambda_ * wi - X.T.dot(y - si)
  w_next = wi - left_side.dot(right_side)
  return w_next

# Define data
X = np.array(
    [[0.2, 3.1, 1],
     [1.0, 3.0, 1],
     [-0.2, 1.2, 1],
     [1.0, 1.1, 1]]
  )
y = np.array([1, 1, 0, 0])
w_0 = np.array([-1, 1, 0])

# for Unregularized rrgression
lambda_ = 0.0

# set printing positions after decimal
np.set_printoptions(precision=4)

# Part A.
s_0 = calc_next_si(X, w_0)
print("Part A. s_0 = %s" % (str(s_0)))

# Part B
w_1 = calc_next_wi(X, w_0, s_0, lambda_)
print("Part B. w_1 = %s" % (str(w_1)))

# Part C
s_1 = calc_next_si(X, w_1)
print("Part C. s_1 = %s" % (str(s_1)))

# Part D
w_2 = calc_next_wi(X, w_1, s_1, lambda_)
print("Part D. w_2 = %s" % (str(w_2)))
```

[**RUBRIC**: (+0.5 point) for correct answer in each sub-part. Answer is correct iff within ±0.0002 of the values below. NO points for code or intermediate calculations.]

[**RUBRIC**: Total (+2 points) for 2.4!]

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

Solution: $\mathbf{s}^{(0)} = [0.9478 \;\; 0.8808 \;\; 0.8022 \;\; 0.5249]$.

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

Solution: $\mathbf{w}^{(1)} = [1.3247 \;\; 3.0499 \;\; -6.8291]$.

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

Solution: $\mathbf{s}^{(1)} = [0.9474 \;\; 0.9746 \;\; 0.0312 \;\; 0.1044]$.

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

Solution: $\mathbf{w}^{(2)} = [1.3659 \;\; 4.1575 \;\; -9.1996]$.

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.

Solution: [**RUBRIC**: Total (+20 points) for Q3.]

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.

Solution: Let X be the design matrix (with the i^{th} row being the sample point $\mathbf{x}_i^\top$). Let $\mathbf{w}^{(t)}$ be the weight iterate at step t.

From question 2.1, the batch gradient descent update law is

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon\left(\lambda\mathbf{w}^{(t)} - X^\top(\mathbf{y} - \mathbf{s}(X\mathbf{w}^{(t)}))\right).$$

[**RUBRIC**: (+1 point) for correct update law. Some students might include the fictitious dimension in this formula to avoid penalizing the bias term, that's also correct. Same for Q3.3.]

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.

Solution: The example code uses $\epsilon = 0.0001$ and $\lambda = 0.1$.

[**RUBRIC**: (+0.5 point) for mentioning ϵ used. (+0.5 point) for mentioning λ used.]

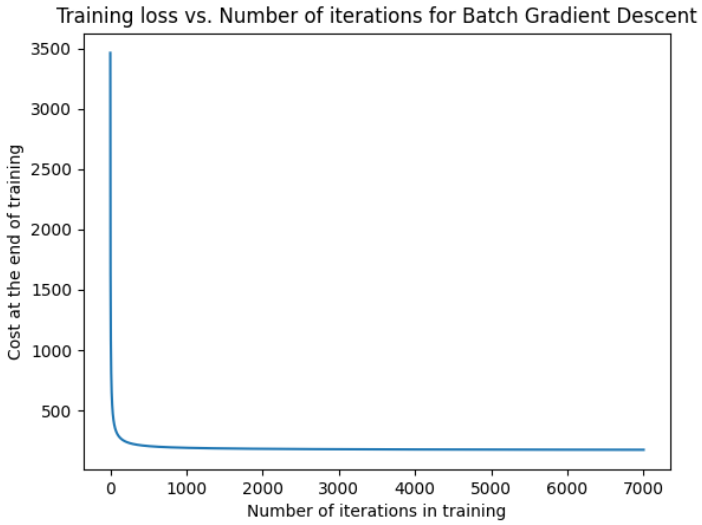
```
# Question 4 Grad Descent

import scipy
from scipy import io
```

```
import numpy as np
import matplotlib.pyplot as plt

training=scipy.io.loadmat('data.mat')
trainfeat=training['X']
trainlabel=training['y']
num_feat=len(trainfeat[0])
num_train=len(trainfeat)
# Appending extra features and labels to training and test sets
extra_feat=np.ones((num_train,1))
train_appended=np.append(trainfeat,extra_feat,axis=1)
train_appended=np.append(train_appended,trainlabel,axis=1)
num_feat_tot=num_feat+1

# Validation set
np.random.seed(0)
np.random.shuffle(train_appended)
vali_size=1000
validation_set=train_appended[:vali_size]
train_set=train_appended[vali_size:num_train]

# Normalizing all the features
means=[np.mean(train_set[:,i]) for i in np.arange(num_feat)]
stds=[np.std(train_set[:,i]) for i in np.arange(num_feat)]
for i in np.arange(num_feat):
    train_set[:,i]=train_set[:,i] - means[i]
    train_set[:,i]=train_set[:,i]/stds[i]

for i in np.arange(num_feat):
    validation_set[:,i]=validation_set[:,i] - means[i]
    validation_set[:,i]=validation_set[:,i]/stds[i]

# Log reg function
def logis(w,X):
    num_samp=X.shape[0]
    s=np.zeros((num_samp,))
    for i in np.arange(num_samp):
        s[i]=np.true_divide(1,1+np.exp(-np.dot(X[i],w)))
    return s

# Batch grad descent on training set
w=np.zeros((num_feat_tot,))
gradJ=np.zeros((num_feat_tot,))

# learning rate setting
learn_step=0.0001

# regularization parameter setting
regu_const=0.1

num_iter=7000
cost=np.zeros((num_iter+1,))
s=logis(w,train_set[:,:num_feat_tot])
cost[0]=-np.dot(train_set[:,num_feat_tot],np.log(s))-np.dot(
    (1-train_set[:,num_feat_tot]),
    np.log(1-s)) + (regu_const/2)* np.sum(np.square(w))

for ite in np.arange(num_iter):
    diff=train_set[:,num_feat_tot]-s
    gradJ=regu_const*w - np.dot(
        np.transpose(train_set[:,:num_feat_tot]),diff)
    w=w-learn_step*gradJ
    s=logis(w,train_set[:,:num_feat_tot])
    cost[ite+1]=-np.dot(train_set[:,num_feat_tot],np.log(s))-np.dot(
        (1-train_set[:,num_feat_tot]),
        np.log(1-s)) + (regu_const/2)*np.sum(np.square(w))
```

```
# Plotting cost vs. iterations
plt.plot(np.arange(num_iter+1),cost)
plt.xlabel('Number of iterations in training')
plt.ylabel('Cost at the end of training')
plt.title('Training loss vs. Number of iterations for Batch Gradient Descent')
plt.savefig('Wine_GD.png')
plt.show()

# Checking on validation set
s_test=logis(w,validation_set[:,:num_feat_tot])
diffe=np.rint(s_test)-validation_set[:,num_feat_tot]
accu=(np.true_divide(
        diffe.size-np.count_nonzero(diffe),vali_size))*100
print(np.rint(s_test).sum())
print("Validation Accuracy is %.2f%%" % (accu))
```

[**RUBRIC**: (+2 points) for an admissible code. A code is admissible iff it matches the code submission provided in the Code deliverable, compiles successfully, and its execution doesn't take exorbitant time or memory.]

[**RUBRIC**: (+1 point) for setting a random seed in code, and mentioning seed value used.]



[**RUBRIC**: (+1 point) for an admissible plot of training loss with the number of iterations. A plot is admissible iff it has a correct title and its axes have axis titles and labels.]

[**RUBRIC**: Total (+5 points) for 3.2!]

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.

Solution: Let X be the design matrix (with the i^{th} row being the sample point $\mathbf{x}_i^\top$). Let $\mathbf{w}^{(t)}$ be the weight iterate at step t.

SGD selects a sample point at random and updates the weight according to it, instead of looking at all examples at each iteration. Although the update function is random, the expected weight change *must be the same* as the update function for batch gradient descent update law.

The SGD update law is

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon\left(\lambda\mathbf{w}^{(t)} - n(y_j - s(\mathbf{x}_j \cdot \mathbf{w}^{(t)}))\mathbf{x}_j^\top\right)$$

where $j \sim \text{Uniform}(\{1, 2, \ldots, n\})$.

[**RUBRIC**: (+1 point) for correct update law. The law is correct iff its expectation with respect to the random index (of training example chosen) is equal to that of batch GD. Some students might omit the n coefficient in the gradient term. That is considered correct since it can be compensated for by changing the step size ϵ.]

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.

Solution: The example code uses $\epsilon = 10^{-6}$ and $\lambda = 0.1$.

[**RUBRIC**: (+0.5 point) for mentioning ϵ used. (+0.5 point) for mentioning λ used.]

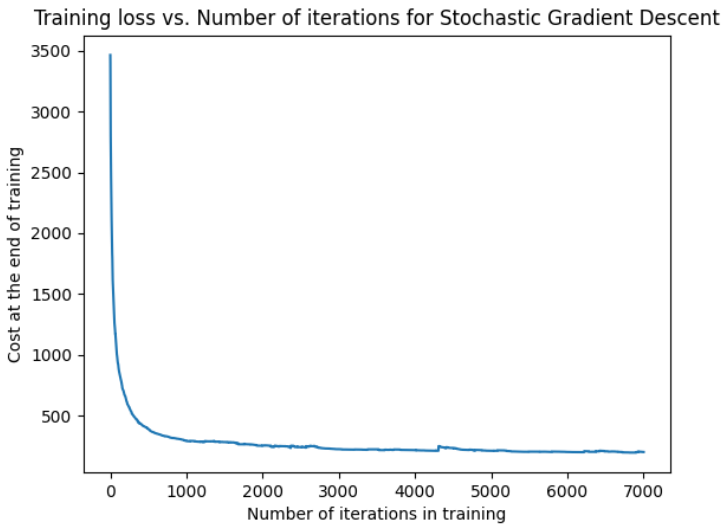
```
# Question 4 Stochastic Grad Descent

import scipy
from scipy import io
import numpy as np
import matplotlib.pyplot as plt

training=scipy.io.loadmat('data.mat')
trainfeat=training['X']
trainlabel=training['y']
num_feat=len(trainfeat[0])
num_train=len(trainfeat)

# Appending extra features and labels to training and test sets
extra_feat=np.ones((num_train,1))
train_appended=np.append(trainfeat,extra_feat,axis=1)
train_appended=np.append(train_appended,trainlabel,axis=1)
num_feat_tot=num_feat+1

# Validation set
np.random.seed(0)
np.random.shuffle(train_appended)
vali_size=1000
validation_set=train_appended[:vali_size]
train_set=train_appended[vali_size:num_train]

# Normalizing all the features
means=[np.mean(train_set[:,i]) for i in np.arange(num_feat)]
stds=[np.std(train_set[:,i]) for i in np.arange(num_feat)]
for i in np.arange(num_feat):
    train_set[:,i]=train_set[:,i] - means[i]
    train_set[:,i]=train_set[:,i]/stds[i]

for i in np.arange(num_feat):
    validation_set[:,i]=validation_set[:,i] - means[i]
    validation_set[:,i]=validation_set[:,i]/stds[i]
```

```
# Log reg function
def logis(w,X):
    num_samp=X.shape[0]
    s=np.zeros((num_samp,))
    for i in np.arange(num_samp):
        s[i]=np.true_divide(1,1+np.exp(-np.dot(X[i],w)))
    return s

# Question 3 SGD with constant learning rate

train_size=num_train-vali_size

# Stochastic grad descent on training set
# For constant learning rate
w=np.zeros((num_feat_tot,))
gradJ=np.zeros((num_feat_tot,))
# For proportional learning rate
w_s=np.zeros((num_feat_tot,))
gradJ_s=np.zeros((num_feat_tot,))


# For constant learning rate, use (for example)
learn_step=1e-6

# for proportional learning rate, use initial (for example)
learn_step_ini=1e-4

# regularization paramter
regu_const=0.1

num_iter=7000


cost=np.zeros((num_iter+1,))    # for constant learning rate
cost_s=np.zeros((num_iter+1,)) # for proportional learning rate

s=logis(w,train_set[:,:num_feat_tot])   # for constant learing rate
ss=logis(w,train_set[:,:num_feat_tot])  # for proportional learning rate

# cost for constant learning rate
cost[0]=-np.dot(train_set[:,num_feat_tot],np.log(s))-np.dot(
    (1-train_set[:,num_feat_tot]),
    np.log(1-s)) + (regu_const/2)*np.sum(np.square(w))
# cost for proportional learning rate
cost_s[0]=-np.dot(train_set[:,num_feat_tot],np.log(ss))-np.dot(
    (1-train_set[:,num_feat_tot]),
    np.log(1-ss)) + (regu_const/2)*np.sum(np.square(w_s))


sample_index=-1
for ite in np.arange(num_iter):
    print(ite)
    learn_step_s=np.true_divide(learn_step_ini,ite+1)
    sample_index=sample_index+1
    if sample_index==train_size:
        np.random.shuffle(train_set)
        sample_index=0

    diff=train_set[:,num_feat_tot]-s    # for constant learning rate
    diff_s=train_set[:,num_feat_tot]-ss   # for proportional learning rate

    # for constant learning rate
    gradJ= regu_const*w - train_size * diff[sample_index]*np.transpose(
        train_set[sample_index,:num_feat_tot])
    # for proportional learning rate
    gradJ_s = regu_const*w_s - train_size * diff_s[sample_index]*np.transpose(
        train_set[sample_index,:num_feat_tot])
```

```
    w=w-learn_step*gradJ    # for constant learning rate
    w_s=w_s-learn_step_s*gradJ_s    # for proportional learning rate

    s=logis(w,train_set[:,:num_feat_tot])   # for constant learning rate
    ss=logis(w_s,train_set[:,:num_feat_tot])    # for proportional learning rate

    # cost for constant learning rate
    cost[ite+1]=-np.dot(train_set[:,num_feat_tot],np.log(s))-np.dot(
        (1-train_set[:,num_feat_tot]),
        np.log(1-s))+(regu_const/2)*np.sum(np.square(w))
    # cost for proportional learning rate
    cost_s[ite+1]=-np.dot(train_set[:,num_feat_tot],np.log(ss))-np.dot(
        (1-train_set[:,num_feat_tot]),
        np.log(1-ss))+(regu_const/2)*np.sum(np.square(w_s))

# Plotting cost vs. iterations
plt.plot(np.arange(num_iter+1),cost)
plt.xlabel('Number of iterations in training')
plt.ylabel('Cost at the end of training')
plt.title('Training loss vs. Number of iterations for Stochastic Gradient Descent')
plt.savefig('Wine_SGD.png')
plt.clf()
plt.close()

# Plotting cost vs. iterations
plt.plot(np.arange(num_iter+1), cost, "-r", label="no decay")
plt.plot(np.arange(num_iter+1),cost_s, "-b", label="decay")
plt.xlabel('Number of iterations in training')
plt.ylabel('Cost at the end of training')
plt.legend(loc="upper left")
plt.title('SGD Training loss vs. iterations with decaying & const learning rate')
plt.savefig('Wine_SGD_combined.png')
plt.clf()
plt.close()

# Checking on validation set
s_test=logis(w,validation_set[:,:num_feat_tot])
ss_test=logis(w_s,validation_set[:,:num_feat_tot])
diffe=np.rint(s_test)-validation_set[:,num_feat_tot]
diffe_s=np.rint(ss_test)-validation_set[:,num_feat_tot]
accu=(np.true_divide(diffe.size-np.count_nonzero(diffe),vali_size))*100
accu_s=(np.true_divide(diffe_s.size-np.count_nonzero(diffe_s),vali_size))*100
print("SGD Validation Accuracy (constant learning rate) is %.2f%%" % (accu))
print("SGD Validation Accuracy (decaying learning rate) is %.2f%%" % (accu_s))
```

[**RUBRIC**: (+2 points) for an admissible code. A code is admissible iff it matches the code submission provided in the Code deliverable, compiles successfully, and its execution doesn't take exorbitant time or memory.]

[**RUBRIC**: (+1 point) for setting a random seed in code, and mentioning seed value used.]



[**RUBRIC**: (+1 point) for an admissible plot of training loss with number of iterations. A plot is admissible iff it has a correct title and its axes have axis titles and labels.]

SGD converges to a final training loss much higher than batch gradient descent.

[**RUBRIC**: Total (+1 points) for observing SGD is slower or converges to a higher cost.]

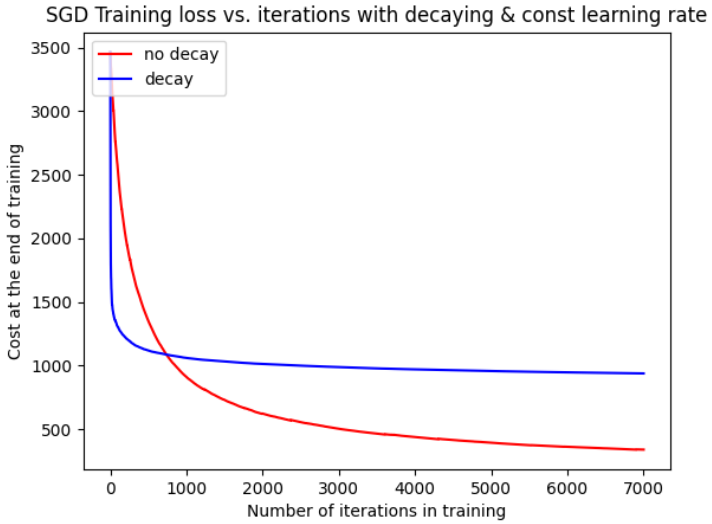
[**RUBRIC**: Total (+6 points) for 3.4!]

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?

Solution: The example code chooses $\delta = 10^{-4}$. There is dramatic improvement in the convergence at the start, but in the end it is not as good. Note that the trend depends on the learning rates chosen in both cases.

[**RUBRIC**: (+1 point) for mentioning value of δ used.]



[**RUBRIC**: (+1 point) for an admissible plot of training loss with the number of iterations. A plot is admissible iff it has a correct title and its axes have axis titles and labels.]

[**RUBRIC**: (+1 point) for stating the trend comparison between decayed and constant learning rates. Any observation in the comparison is acceptable, because it depends on the learning rate choices.]

[**RUBRIC**: Total (+3 points) for 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.

Solution: [**RUBRIC**: For 3.6, give ZERO points if Kaggle username is not mentioned.]

[**RUBRIC**: (+0.5 point) for mentioning best Kaggle score. (+0.5 point) for a short description of the best algorithm.]

[**RUBRIC**: (+3 point) if Kaggle test accuracy is equal or better than 90%.]

[**RUBRIC**: Total (+4 points) for 3.6!]

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."*

Solution: [**RUBRIC**: Total (+4 points) for Q4.]

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.

   Solution:

   By Bayes' Theorem,

   $$f(\mathbf{w}|(\mathbf{x}_i, y_i)_{i\in[n]}) = \frac{f(\{y_i\}_{i\in[n]}|\mathbf{w}, \{\mathbf{x}_i\}_{i\in[n]})f(\mathbf{w})}{f(\{y_i\}_{i\in[n]}|\{\mathbf{x}_i\}_{i\in[n]})}. \tag{4}$$

   Here $\{y_i\}_{i\in[n]}$ is the collection of n random variables y_i, and similarly $\{\mathbf{x}_i\}_{i\in[n]}$.

   [**RUBRIC**: (+1 point) for the correct application of Bayes' Theorem.]

   By the independence of the y_i's,

   $$f(\{y_i\}_{i\in[n]}|\mathbf{w}, \{\mathbf{x}_i\}_{i\in[n]}) = \prod_{i\in[n]} f(y_i|\mathbf{w}, \mathbf{x}_i) = (\sigma\sqrt{2\pi})^{-n} \exp\left(-\sum_{i=1}^{n} \frac{(y_i - \mathbf{w}\cdot\mathbf{x}_i)^2}{2\sigma^2}\right).$$

   Similarly, by the independence of the w_j's,

   $$f(\mathbf{w}) = \prod_{j\in[d]} f(w_j) = (2b)^{-d} e^{-\sum_{j=1}^{d} |w_j|/b}.$$

   [**RUBRIC**: (+0.5 point) for correct expression using the independence of the y_i's. (+0.5 point) for correct expression using the independence of the w_j's.]

Next, observe that the denominator of Equation (4) is $f(\{y_i\}_{i\in[n]}|\{\mathbf{x}_i\}_{i\in[n]}) = \mathbb{E}_{\mathbf{w}}[f(\{y_i\}_{i\in[n]}|\mathbf{w}, \{\mathbf{x}_i\}_{i\in[n]}]$ which is a constant. Therefore,

$$f(\mathbf{w}|(\mathbf{x}_i, y_i)_{i\in[n]}) \propto \prod_{i\in[n]} f(y_i|\mathbf{w}, \mathbf{x}_i) \prod_{j\in[d]} f(w_j) \propto e^{-\sum_{i=1}^{n}(y_i-\mathbf{w}\cdot\mathbf{x}_i)^2/(2\sigma^2)} e^{-\sum_{j=1}^{d}|w_j|/b}$$

[**RUBRIC**: (+1 point) for correct expression of posterior upto a proportionality constant.]

[**RUBRIC**: Total (+3 points) for 4.1!]

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.

Solution: From question 4.1 we have $\ell(\mathbf{w}) = -\frac{\sum_{i=1}^{n}(y_i-\mathbf{w}\cdot\mathbf{x}_i)^2}{2\sigma^2} - \frac{\|\mathbf{w}\|_1}{b} + \ln c$ where c is the proportionality constant. Clearly, maximizing $\ell(\cdot)$ (ignore $\ln c$ as it doesn't affect the objective) is equivalent to minimizing $\sum_{i=1}^{n}(y_i - \mathbf{w}\cdot\mathbf{x}_i)^2 + \frac{2\sigma^2}{b}\|\mathbf{w}\|_1$

[**RUBRIC**: (+0.5 point) for correct expression of $\ell(\mathbf{w})$ within a constant. (+0.5 point) for correct value of $\lambda = \frac{2\sigma^2}{b}$.]

[**RUBRIC**: Total (+1 point) for 4.2!]

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

Solution: [**RUBRIC**: Total (+10.5 points) for Q 5.]

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.

   Solution:

```
import numpy as np
import matplotlib.pyplot as plt

def lp_norm(X, Y, p):
    return (np.abs(X)**p + np.abs(Y)**p)**(1/p)

x = np.linspace(-2, 2, 500)
y = np.linspace(-2, 2, 500)
X, Y = np.meshgrid(x, y)

# subpart (a)
Z = lp_norm(X, Y, 0.5)
plt.contour(X, Y, Z)
plt.show()

# subpart (b)
Z = lp_norm(X, Y, 1)
plt.contour(X, Y, Z)
plt.show()


# subpart (c)
Z = lp_norm(X, Y, 2)
plt.contour(X, Y, Z)
plt.show()
```

   [**RUBRIC**: (+0.5 point) for each correct subpart, 1.5 points in total]

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.

   Solution: Expand the objective and use $X^\top X = nI$ to get $J_1(\mathbf{w}) = \mathbf{w}^\top X^\top X\mathbf{w} - 2\mathbf{y}^\top X\mathbf{w} + \|\mathbf{y}\|^2 + \lambda\|\mathbf{w}\|_1 = \|\mathbf{y}\|^2 + n\|\mathbf{w}\|^2 - 2\mathbf{y}^\top X\mathbf{w} + \lambda\|\mathbf{w}\|_1$. Now substitute $n\|\mathbf{w}\|^2 = \sum_{j\in[d]} nw_j^2$; $\lambda\|\mathbf{w}\| = \lambda\sum_{j\in[d]}\|w_j\|$; $2\mathbf{y}^\top X\mathbf{w} = 2\sum_{j\in[d]}(\mathbf{y}\cdot\mathbf{x}_{*j})w_j$ and simplify to get $f(\mathbf{u}, a) = na^2 - 2(\mathbf{y}\cdot\mathbf{u})a + \lambda\|a\|$.

   [**RUBRIC**: (+1 point) for correct expression for $f(\mathbf{u}, a)$.]

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 $w_i^* < 0$.

Solution:

From question 5.2 we know that to find component w_i^* only $g_i(u) \triangleq nu^2 - 2k_iu + \lambda||u||$ needs to be minimized (here $k_i \triangleq \mathbf{y} \cdot \mathbf{x}_{*i}$). Using $\mathbb{I}[\dots]$ for indicator variables, we have

$$g_i(u) = \mathbb{I}[u > 0](nu^2 - 2k_iu + \lambda u) + \mathbb{I}[u < 0](nu^2 - 2k_iu - \lambda u).$$

Necessary and Sufficient condition for $w_i^* > 0$. Notice that $g_i(0) = 0$, and $w_i^* > 0$ iff $\inf_{u>0} g_i(u) < \inf_{u\leq 0} g_i(u)$. Use properties of quadratic functions to see $\inf_{u>0} g_i(u) = -(1/n)(k_i - \lambda/2)^2\mathbb{I}[(k_i - \lambda/2)/n > 0]$ and $\inf_{u\leq 0} g_i(u) = -(1/n)(k_i + \lambda/2)^2\mathbb{I}[(k_i + \lambda/2)/n \leq 0]$.

Thus, $w_i^* > 0$ iff $-(1/n)(k_i - \lambda/2)^2\mathbb{I}[(k_i - \lambda/2)/n > 0] < -(1/n)(k_i + \lambda/2)^2\mathbb{I}[(k_i + \lambda/2)/n \leq 0]$, which in turn is true iff $k_i - \lambda/2 > 0$.

Necessary and Sufficient condition for $w_i^* < 0$. Similarly, $w_i^* < 0$ iff $\inf_{u<0} g_i(u) < \inf_{u\geq 0} g_i(u)$. Thus, $w_i^* < 0$ iff $-(1/n)(k_i + \lambda/2)^2\mathbb{I}[(k_i + \lambda/2)/n < 0] < -(1/n)(k_i - \lambda/2)^2\mathbb{I}[(k_i - \lambda/2)/n \geq 0]$, which in turn is true iff $k_i + \lambda/2 < 0$.

Therefore, by negation of $(w_i^* > 0) \cup (w_i^* < 0)$, deduce that $w_i = 0$ iff $-\lambda/2 \leq k_i \leq \lambda/2$.

[**RUBRIC**: (+0.5 point) for correct necessary and sufficient condition of each of three cases: $w_i^* < 0$, $w_i^* > 0$, $w_i^* = 0$. Total (+1.5 points) when all 3 conditions are correct. *Make sure that you distinguish* $<$ and $\leq$ when judging if student's condition is correct. Similarly $>$ is different from $\geq$. Do NOT accept the wrong inequality sign.]

[**RUBRIC**: Students need to only derive the necessary and sufficient condition for two of the three conditions; the third is immediately obtained by negation. Do NOT award any points for proof of necessary and sufficiency of the third condition.

Thus, for each of *at least* 2 of the 3 conditions supplied by the student:

- (+1 point) for correct explanation that condition is necessary. Explanation for wrong condition (including wrong inequality) is completely wrong.
- (+1 point) for correct explanation that condition is sufficient. Explanation for wrong condition (including wrong inequality) is completely wrong.

Total points for necessary and sufficiency are: 2 conditions $\times$ (1 + 1) points = (+4 points).]

[**RUBRIC**: Total (1.5 + 4 = +5.5 points) for 5.3!]

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 $\mathbf{w}^\#$.

   Solution: $\nabla_{\mathbf{w}} J_2 = 2n\mathbf{w}^\# - 2X^\top\mathbf{y} + 2\lambda\mathbf{w}^\# = 0$. Thus, $w_i^\# = \dfrac{\mathbf{y} \cdot \mathbf{x}_{*i}}{n + \lambda}$ which is 0 iff $\mathbf{y} \cdot \mathbf{x}_{*i} = 0$

   [**RUBRIC**: (+0.5 point) for correct expression of $\nabla_{\mathbf{w}} J_2$. (+0.5 point) for correct condition.]

   [**RUBRIC**: Total (+1 point) for 5.4!]

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?

Solution: $w_i^{\#} = 0$ iff $\mathbf{y} \cdot \mathbf{x}_{*i} = 0$ whereas $w_i^* = 0$ iff $-\lambda/2 \leq \mathbf{y} \cdot \mathbf{x}_{*i} \leq \lambda/2$. Since $\mathbf{y} \cdot \mathbf{x}_{*i}$ is more likely to lie in an interval than be a specific value, $\mathbf{w}^*$ is more likely to be sparse.

[**RUBRIC**: (+0.5 point) for saying that $\mathbf{w}^*$ (corresponding to ℓ_1 regularization) is more likely to be sparse. (+1 point) for correct argument why it is so.]

[**RUBRIC**: Total (+1.5 points) for 5.5!]

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.